

TOWARDS AN UNDETECTABLE COMPUTER VIRUS

A Project Report

Presented to

The Faculty of the Department of Computer Science

San Jose State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Computer Science

by

Priti Desai

December 2008

© 2008

Priti Desai

ALL RIGHTS RESERVED

SAN JOSE STATE UNIVERSITY

The Undersigned Project Committee Approves the Project Titled

TOWARDS AN UNDETECTABLE COMPUTER VIRUS

by
Priti Desai

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

Dr. Mark Stamp	Department of Computer Science	Date
----------------	--------------------------------	------

Dr. Robert Chun	Department of Computer Science	Date
-----------------	--------------------------------	------

Mr. Vijay Seshadri	Symantec Antivirus Company	Date
--------------------	----------------------------	------

APPROVED FOR THE UNIVERSITY

Associate Dean	Office of Graduate Studies and Research	Date
----------------	---	------

ABSTRACT

TOWARDS AN UNDETECTABLE COMPUTER VIRUS

by Priti Desai

Metamorphic viruses modify their own code to produce viral copies which are syntactically different from their parents. The viral copies have the same functionality as the parent but may have different signatures. This makes signature-based virus scanners unreliable for detecting metamorphic viruses. But statistical pattern analysis tool such as Hidden Markov Models (HMMs) can detect metamorphic viruses.

Virus writers use many different code obfuscation techniques to generate metamorphic viruses. In this project we develop a metamorphic engine using code obfuscation techniques. Our metamorphic engine is designed to produce highly diverse morphed copies of the base virus. We show that commercial virus scanners cannot detect metamorphic viruses produced by our engine. We then proceed to determine whether HMMs can detect metamorphic viruses generated by our engine.

ACKNOWLEDGEMENTS

I would like to thank Dr. Mark Stamp for trusting me with his idea. A special thank to Dr. Stamp for his guidance, encouragement, and support throughout the project.

This project would have not been possible without a special support of my loving husband Mrugesh. I would like to thank Mrugesh for his encouragement, patience and help throughout the process, especially for those sleepless nights accompanying me.

TABLE OF CONTENTS

1. INTRODUCTION	1
2. COMPUTER VIRUS.....	2
3. ANTIVIRUS DEFENSE TECHNIQUES.....	3
3.1 SIGNATURE DETECTION.....	3
3.2 HEURISTIC ANALYSIS.....	3
4. ADVANCED CODE EVOLUTION TECHNIQUES.....	4
4.1 ENCRYPTION.....	4
4.2 POLYMORPHISM.....	4
4.3 METAMORPHISM	4
4.3.1 Anatomy of a Metamorphic Virus.....	5
4.3.2 The Metamorphic Virus According to a Virus Writer	6
5. CODE OBFUSCATION TECHNIQUES.....	7
5.1 REGISTER USAGE EXCHANGE (REGISTER RENAMING).....	7
5.2 DEAD CODE INSERTION.....	8
5.3 SUBROUTINE PERMUTATION.....	9
5.4 EQUIVALENT CODE SUBSTITUTION	10
5.5 TRANSPOSITION.....	10
5.6 CHANGING THE CONTROL FLOW (CODE REORDERING THROUGH JUMPS).....	11
5.7 SUBROUTINE INLINING AND SUBROUTINE OUTLINING.....	11
6. SIMILARITY TEST.....	13
7. HIDDEN MARKOV MODEL.....	14
7.1 HMM AS VIRUS DETECTION TOOL.....	17
8. IMPLEMENTATION.....	19
8.1 INTRODUCTION.....	19
8.2 GOALS.....	20
8.3 CODE OBFUSCATION TECHNIQUES USED.....	20
8.3.1 Dead Code Insertion.....	20
8.3.2 Equivalent instruction substitution.....	24
8.3.3 Transpose.....	24
9. EXPERIMENTS.....	26
9.1 COMMERCIAL VIRUS SCANNER	27
9.2 SIMILARITY TEST.....	27
9.3 HMM.....	29
9.3.1 N generation viruses against the base virus model.....	29
9.3.2 The Base virus against the morphed virus model.....	30
9.3.3 Normal files against 9th generation virus model.....	31

<i>9.3.4 Morphed viruses against normal file model</i>	33
10. CONCLUSION	34
11. FUTURE WORK	34
REFERENCES	36
APPENDIX B: EQUIVALENT INSTRUCTION SUBSTITUTION	39
APPENDIX C: SIMILARITY TESTS	43
APPENDIX D: HIDDEN MARKOV MODEL OF THE BASE VIRUS	47
APPENDIX E: HIDDEN MARKOV MODELS OF NORMAL FILES	49
APPENDIX F: HIDDEN MARKOV MODEL OF 9TH GENERATION VIRUSES	51

LIST OF FIGURES

FIGURE 1: PSEUDO CODE OF A COMPUTER VIRUS [12].....	2
FIGURE 2: PSEUDO CODE OF INFECT MODULE [12].....	3
FIGURE 3: METAMORPHIC VIRUS GENERATIONS.....	5
FIGURE 4: ANATOMY OF A METAMORPHIC ENGINE [15].....	5
FIGURE 5: TWO DIFFERENT GENERATIONS OF REGSWAP [4].....	8
FIGURE 6: DEAD CODE INSERTION IN EVOL VIRUS [8].....	9
FIGURE 7: SUBROUTINE PERMUTATION [4].....	9
FIGURE 8: EXAMPLE OF CONTROL FLOW MODIFICATION [19].....	11
FIGURE 9: SUBROUTINE INLINING.....	12
FIGURE 10: SUBROUTINE OUTLINING.....	12
FIGURE 11: SIMILARITY GRAPH.....	14
FIGURE 12: TEMPERATURE TRANSITION PROBABILITY.....	14
FIGURE 13: TREE SIZE PROBABILITY.....	15
FIGURE 14: HMM MODEL.....	16
FIGURE 15: TRAINING DATA	18
FIGURE 16: HMM MODEL.....	18
FIGURE 17: THE RESULT FILE.....	19
FIGURE 18: BASE VIRUS OPCODES AND THEIR FREQUENCY.....	20
FIGURE 19: OPCODES OF NORMAL FILE AND THEIR FREQUENCY.....	21
FIGURE 20: ALGORITHM TO INSERT NOP SEQUENCE ON ENTRY POINT.....	22
FIGURE 21: ALGORITHM TO INSERT RANDOM NOP SEQUENCE.....	22
FIGURE 22: ALGORITHM FOR TRANSPOSE.....	25
FIGURE 23: HIGH LEVEL ALGORITHM OF METAMORPHIC ENGINE.....	26
FIGURE 24: OVER ALL PROCESS.....	27
FIGURE 25: SIMILARITY RESULTS OF THE BASE VIRUS V/S 9 DIFFERENT GENERATIONS.....	28
FIGURE 26: GRAPH OF SIMILARITY OF TWO N GENERATIONS.....	29
FIGURE 27: N (1-9) GENERATION VIRUSES TESTED AGAINST BASE VIRUS MODEL.....	30
FIGURE 28: BASE VIRUS TESTED AGAINST N GENERATION MODELS.....	31
FIGURE 29: FAMILY VIRUSES AND NORMAL FILES TESTED AGAINST 9TH GENERATION MODEL	32

FIGURE 30: FAMILY VIRUSES AND 9TH GENERATION VIRUSES TESTED AGAINST NORMAL MODEL.....33
FIGURE 31: CHANGE IN FILE SIZES OVER 9 GENERATIONS.....35

LIST OF TABLES

TABLE 1: METAMORPHIC VIRUSES AND CODE OBFUSCATION TECHNIQUES [19].....	7
TABLE 2: EXAMPLES OF INSTRUCTION SUBSTITUTION USED BY W32/METAPHOR VIRUS [19].....	10
TABLE 3: PROBABILITIES OF OBSERVING (S, M, S, L) FOR ALL POSSIBLE STATE SEQUENCES.....	17
TABLE 4: ARITHMETIC DEAD CODE INSTRUCTIONS.....	21
TABLE 5: EVOL TRANSFORMATIONS [6].....	23
TABLE 6: SUBSTITUTIONS FOR ADD.....	24
TABLE 7: HMM OF BASE VIRUS TESTED WITH 9 GENERATIONS.....	29
TABLE 8: THE BASE VIRUS TESTED AGAINST N GENERATION MODEL.....	30
TABLE 9: RESULTS OF 9TH GENERATION VIRUSES TESTED AGAINST 9TH GENERATION MODEL.....	32
TABLE 10: RESULTS OF 9TH GENERATION VIRUSES TESTED AGAINST NORMAL MODEL.....	33

1. Introduction

A computer virus is a malware that, when executed, tries to infect other executables and alter their default behavior [12]. A virus copies itself into an infected executable without permission or knowledge of a user [13]. According to Fred Cohen, "A computer virus is a program that can infect other programs by modifying them to include a possibly evolved copy of itself" [17]. The first computer virus was a boot sector virus called Brain, created in 1986 by two brothers, Basit and Amjad Farooq Alvi, operating out of Lahore, Pakistan.

Generally a computer virus causes damage to the host machine. The damage can be done to a number of different components of the computer's operating and file system. These include system sectors, files, macros, companion files and source code. The always connected world of internet is a soft target for viruses. Viruses use internet connectivity to spread across the world faster and create havoc. The early detection of viruses is imperative to minimize the damages caused by them.

There are many antivirus defense mechanisms available today. These include signature detection and code emulation. The signature based virus detection tools search all the files on a system for a signature. Code emulation creates a virtual machine and executes a virus on the virtual machine for detection. Once the virus is detected, it is no longer a threat.

To bypass signature detection technique, virus writers have to create new viruses or change the existing viruses. Virus writers evade signature detection by generating metamorphic copies of a virus. Metamorphic viruses change their appearance while keeping the same functionality. Metamorphic viruses use different code obfuscation techniques to change the structure of the code. These techniques include code reordering through jumps, subroutine permutation, dead code insertion, equivalent instruction substitution, and rearrangement of instruction order (transposition).

The statistical pattern analysis is the most successful technique to detect metamorphic viruses [2]. Hidden Markov Model (HMM) is the well known statistical pattern analysis tool. HMM has been widely used in speech recognition and protein modeling. HMM has been extended to detect metamorphic viruses.

Metamorphic viruses with combination of code reordering through jumps and dead code insertion evades signature detection but are detected by HMM [9]. In this project we determine whether extensive metamorphism can evade HMM.

The aim of this project is to develop a metamorphic engine. We used code obfuscation techniques like equivalent instruction substitution, dead code insertion and rearrangement of instruction order. We designed our metamorphic engine to generate highly discrete copies of the base virus. These morphed copies are tested against the HMM model of the

base virus family, normal files, and our own morphed copies. We also tested our morphed copies against commercial virus scanners.

This paper is organized as follows:

- Section 2 contains information about computer viruses.
- Section 3 discusses various anti-virus technologies currently used.
- Section 4 contains information about the evolution of viruses.
- Section 5 details a few code obfuscation techniques that are used for generating metamorphic variants.
- Section 6 describes our virus similarity test.
- Section 7 introduces HMM as virus detection tool.
- Section 8 and 9 details the design, implementation, and experimental results of our metamorphic engine.
- Section 10 draws conclusions based upon our findings.
- Section 11 discusses additional future enhancements.

2. Computer Virus

“A computer virus is a malicious program that modifies other host files to replicate. The host is modified to include a complete copy of the malicious code program. The execution of the infected host file infects other objects” [16]. Generally a computer virus consists of three modules [12].

```
def virus() :  
    infect ()  
    if trigger () is true then  
        payload ()
```

Figure 1: Pseudo code of a computer virus [12]

Infect defines how a virus spreads. One common infection mechanism is to modify host to contain copy of virus code. *Trigger* is a test to decide to deliver the payload or not. *Payload* defines damage done by the virus. Trigger and payload are optional. Figure 1 shows pseudo code of a virus.

```
def infect() :
    repeat k times:
        target = select_target()
        if no target then
            return
        infect_code (target)
```

Figure 2: Pseudo code of infect module [12]

Infect module selects a target to infect. Generally *k* targets are selected on each run. *Select_target* defines criteria by which a target is selected. The same target should not be selected repeatedly otherwise infecting the same code repeatedly may reveal the presence of the virus. *infect_code* performs actual infection by inserting virus's code into the target.

3. Antivirus Defense Techniques

This section presents some of the most popular techniques used by antivirus software to detect computer viruses.

3.1 Signature Detection

A signature is a string of bits found in a virus [1]. An effective signature is the string of bits which is commonly found in viruses but not likely to be found in normal programs. Generally each virus has its own unique signature. All known signatures are organized in a database. A signature-based virus detection tool searches for a known signature in all the files on a system. The following example is a signature of W32/Beast virus in infected executable files [22].

83EB 0274 EB0E 740A 81EB 0301 0000

The virus scanner searches executables for this signature. If this signature is present in any executable file, it is declared as the Beast virus.

3.2 Heuristic Analysis

Heuristic analysis is useful in detecting new or unknown viruses. Heuristic analysis can be static or dynamic. Static heuristics mainly analyzes the file format and the code structure of virus body. Dynamic heuristics use code emulators to detect unusual behavior while the virus code is running inside the emulator. The following examples are the suspicious characteristics of heuristic analysis of 32 bit windows viruses [4]:

- Code execution starts in the last section

- Virtual size is incorrect in PE header
- Possible “Gap” between sections
- Suspicious code section name
- Suspicious imports from Kernel32.dll by ordinal

Heuristic analysis creates many false positives. A false positive is to declare a benign program as a virus. An antivirus scanner creating many false positives loses user’s trust and interest. The following section explains techniques used by virus writers to evade signature detection and heuristic analysis.

4. Advanced Code Evolution Techniques

To bypass detection by the user or antivirus software, viruses use different concealment strategies. Some of the concealment strategies are listed below.

4.1 Encryption

Encryption is the simplest way to hide virus body. Encryption changes the appearance of a virus. An encrypted virus consists of a small decrypting module (a decryptor) and an encrypted virus body. Generally simple encryption methods are used like XOR of the key with each byte of the virus body. And if a different key is used for each infection, the encrypted virus body will look different. But the decryptor always remains constant. As a result, detection is still possible. A virus scanner can recognize the decryptor in most cases.

4.2 Polymorphism

To overcome drawbacks of encryption, polymorphic virus mutates virus body along with decryptor. Polymorphic virus has no part that stays constant on each infection. To detect polymorphic viruses, antivirus software implements a code emulator which emulates the decryption process and dynamically decrypts the encrypted virus body. Polymorphic viruses after decryption have a constant virus body. Therefore decrypted virus body can be easily detected.

4.3 Metamorphism

Unlike polymorphic viruses, metamorphic viruses do not employ encryption. Metamorphic viruses change the appearance of the code while keeping the functionality of virus intact. Metamorphic viruses use several code obfuscation techniques including Instruction reordering, data reordering, subroutine inlining, subroutine outlining, register renaming, code permutation, instruction substitution, and garbage code insertion. Figure 3 shows the distinct signatures of the metamorphic viruses.

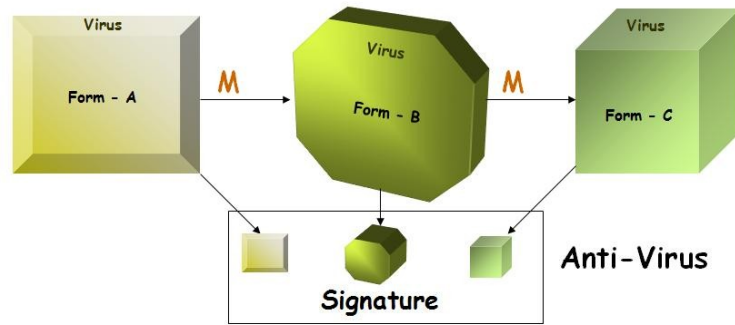


Figure 3: Metamorphic virus generations

4.3.1 Anatomy of a Metamorphic Virus

Generally a metamorphic virus has the metamorphic engine embedded within itself. During infection a metamorphic virus creates morphed copy of itself using the embedded engine. A typical metamorphic engine consists of following functional units. Some of these units are optional.

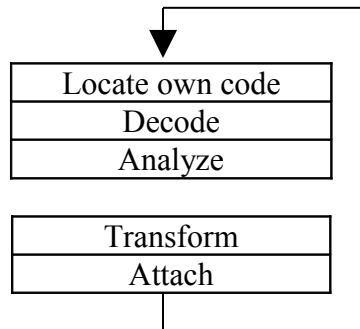


Figure 4: Anatomy of a metamorphic engine [15]

A metamorphic engine reads in the virus executable and locates the code to be transformed using *locate own code* module. Every engine has its own transformation rules. The transformation rules define how a particular opcode or a sequence of opcodes is to be transformed. *Decode* module extracts these rules by disassembling. *Analyze* module analyzes current copy of the virus and determines the transformations to be applied for generating next morphed copy. *Transform* module performs the actual transformations. It replaces an instruction or block of instructions with the other equivalent code. The last module *attach* attaches the transformed copy to a host.

4.3.2 *The Metamorphic Virus According to a Virus Writer*

Generally a virus writer considers how to infect a file and the behavior of the infected file. In addition to these, a virus writer writing a metamorphic virus has to consider how to generate morphed copies of the virus. To generate morphed copies, a metamorphic engine is embedded within the virus body. A typical metamorphic engine may contain [18]:

1. Internal disassembler
2. Opcode shrinker
3. Opcode expander
4. Opcode swapper
5. Relocator/recalculator
6. Garbager
7. Cleaner

Internal disassembler disassembles the binary / executable code, instruction by instruction. *Opcode shrinker* performs optimization of instructions. *Opcode shrinker* replaces two or more instructions with one equivalent instruction. *Opcode expander* is the reverse operation of opcode shrinker. It replaces one instruction with several instructions. *Opcode swapper* changes the order of the instructions. Generally it swaps two unrelated instructions. *Relocator* relocates relative references like jump and call. *Garbager* inserts do-nothing instructions. *Cleaner* undoes Garbager, i.e. it removes do-nothing instructions inserted by Garbager.

Characteristics of an effective metamorphic engine [18]:

1. A metamorphic engine should be able to handle any opcode of an assembly language. An engine should know all of the opcodes.
2. Opcode shrinker and swapper should process more than one instruction concurrently.
3. Use Garbager in moderate amount.
4. Garbage should not affect actual instructions.
5. Opcode swapper should analyze each instruction and should not affect the execution of next instruction.

We have implemented the metamorphic engine as an external tool. This tool reads in a hand written assembly program or disassembled virus executable.

5. Code Obfuscation Techniques

Metamorphic engine uses code obfuscation techniques to produce morphed copies of an original program. Generally the obfuscated code is more difficult to read and understand [1]. Code obfuscation can be used to generate different looking copies of a single parent file. This section explains the code obfuscation techniques for assembly programs.

Code obfuscation techniques for assembly programs operate on both the control flow and data section of the program [19]. Control flow obfuscation involves reordering of instructions through insertion of jumps. Data flow obfuscation can be done in many ways such as equivalent code substitution, subroutine permutation, dead code insertion, register renaming, and transposition. Table 1 summarizes some well known metamorphic viruses and the code obfuscation techniques used by them.

Table 1: Metamorphic Viruses and Code Obfuscation Techniques [19]

	EVOL (2000)	ZMIST (2001)	ZPERM (2000)	REGSWAP (2000)	METAPHOR (2001)
Instruction Substitution				✓	
Instruction Permutation	✓	✓			✓
Dead code Insertion	✓	✓			✓
Variable Substitution	✓	✓		✓	✓
Changing the Control Flow		✓	✓		✓

5.1 Register Usage Exchange (Register Renaming)

Register renaming modifies register operands of an instruction without changing the instruction itself. The instructions remain constant across all morphed copies only the operands change. RegSwap was one of the early metamorphic viruses to use register usage exchange. Figure 5 shows two pieces of code from two different generations of RegSwap.

```

a.)
5A          pop     edx
BF04000000 mov     edi,0004h
8BF5       mov     esi,ebp
B80C000000 mov     eax,000Ch
81C2880000 add     edx,0088h
8B1A       mov     ebx,[edx]
899C8618110000 mov    [esi+eax*4+00001118],ebx

b.)
58          pop     eax
BB04000000 mov     ebx,0004h
8BD5       mov     edx,ebp
BF0C000000 mov     edi,000Ch
81C0880000 add     eax,0088h
8B30       mov     esi,[eax]
89B4BA18110000 mov    [edx+edi*4+00001118],esi

```

Figure 5: Two different generations of RegSwap [4]

Two generations of RegSwap (a) and (b) have the same sequence of instructions but the registers are different. Here the registers `edx`, `edi`, `esi`, `eax`, and `ebx` have been replaced by `eax`, `ebx`, `edx`, `edi`, and `esi` respectively.

5.2 Dead Code Insertion

Inserting dead code or do-nothing instruction does not affect the execution of the original code. Dead code can be a single instruction or a block of instructions. Inserting dead code changes the appearance of a program. Do-nothing instructions such as “move `eax`, `eax`”, “`shl eax, 0`”, “`add ax, 0`”, and “`inc eax`” followed by “`dec eax`” make program look different. Adding new block of dead code on each generation creates different looking programs with the same functionality. The Evol virus had implemented dead code insertion by adding a block of dead code between core instructions as shown in figure 6.

```

C7060F000055  mov [esi], 5500000Fh
C746048BEC5151  mov [esi+0004], 5151EC8Bh

```

```

BF0F00055    mov edi, 5500000Fh
893E         mov [esi], edi
5F          pop edi           ; garbage
52          push edx          ; garbage
B640        mov dh, 40           ; garbage
BA8BEC5151   mov edx, 5151EC8Bh
53          push ebx           ; garbage
8BDA        mov ebx, edx
895E04       mov [esi+0004], ebx

```

Figure 6: Dead code insertion in Evol virus [8]

These two blocks of instructions look different but have the same functionality. The instructions commented garbage does not have any impact on the functionality of the code.

5.3 Subroutine Permutation

This is a simple obfuscation technique in which the subroutines of a program are reordered. A program with n different subroutines can generate $(n-1)!$ different subroutine permutations. Subroutine permutation does not affect the functionality of a program as the order of subroutine is not important for its execution. Figure 3 shows an example of subroutine permutation from [4].

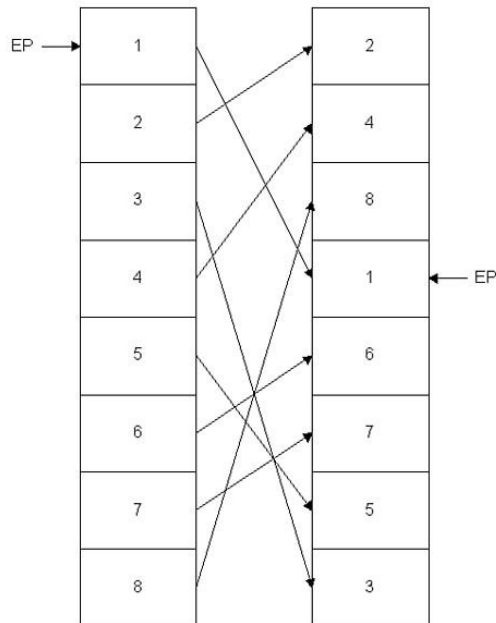


Figure 7: Subroutine permutation [4]

5.4 Equivalent Code Substitution

Equivalent code substitution is the replacement of an instruction with an equivalent instruction or an equivalent block of instructions. In assembly language, generally a task can be achieved in different ways. e.g. “inc eax” is equivalent to “add eax, 1”, “move eax, edx” is equivalent to “push edx” followed by “pop eax” and so on. This property of assembly language where a single task can be implemented in multiple ways is used in equivalent code substitution.

Table 2: Examples of instruction substitution used by W32/MetaPhor virus [19]

Single Instruction	Instruction block
XOR Reg,Reg	MOV Reg,0
MOV Reg,Imm	PUSH Imm POP Reg
OP Reg,Reg2	MOV Mem,Reg OP Mem,Reg2 MOV Reg,Mem

Table 2 shows some examples of equivalent code substitution used by Win32/MetaPhor. “Xor Reg, Reg” is equivalent to moving 0 into the Reg because xor of a value with itself is 0. An equivalent instruction block for “OP Reg, Reg2” uses the ability of a processor to perform the same operation with memory.

5.5 Transposition

Transposition or instruction permutation modifies the instruction execution order in a program. This can be done only if no dependency exists among instructions. Consider two instructions Instruction-1 (op1 R1, R2) and Instruction-2 (op2 R3, R4). These two instructions can be swapped if following conditions are satisfied.

1. R1 is not equal to R3
2. R1 is not equal to R4
3. R2 is not equal to R3

For example, instructions “mov eax, edx” and “add ecx, 5” can be swapped as they satisfy the transpose criteria.

```

...
mov  eax, edx
add  ecx, 5
...

```

↔

```

...
add  ecx, 5
mov  eax, edx
...

```

5.6 Changing the Control Flow (Code Reordering through jumps)

Code reordering inserts conditional or unconditional branching instruction after every instruction or a block of instructions. These blocks defined by the branching instructions are permuted to change the control flow. The modified code is called Spaghetti Code. The conditional branching instruction is always preceded by a test instruction which always forces the execution of the branching instruction.

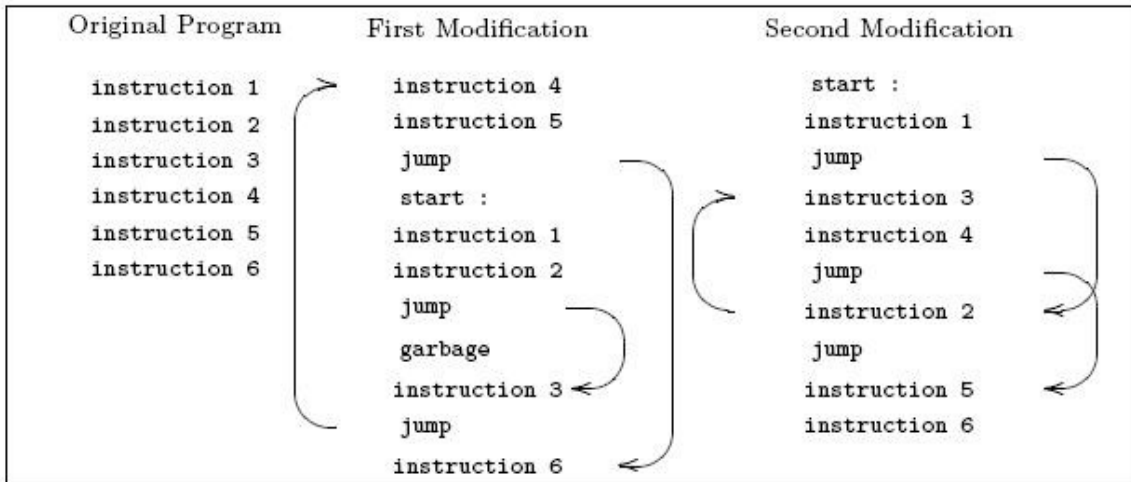


Figure 8: Example of control flow modification [19]

Figure 8 shows an example of spaghetti code. Here, consecutive instructions are permuted and linked together by unconditional jumps. The reordering of instructions does not modify the order in which they are executed.

5.7 Subroutine Inlining and Subroutine Outlining

Subroutine inlining is a technique in which a subroutine call is replaced with its code [CVM]. Subroutine inlining is a code obfuscation technique similar to dead code insertion, the only difference is former inserts subroutine code whereas later inserts arbitrary dead code in a program.

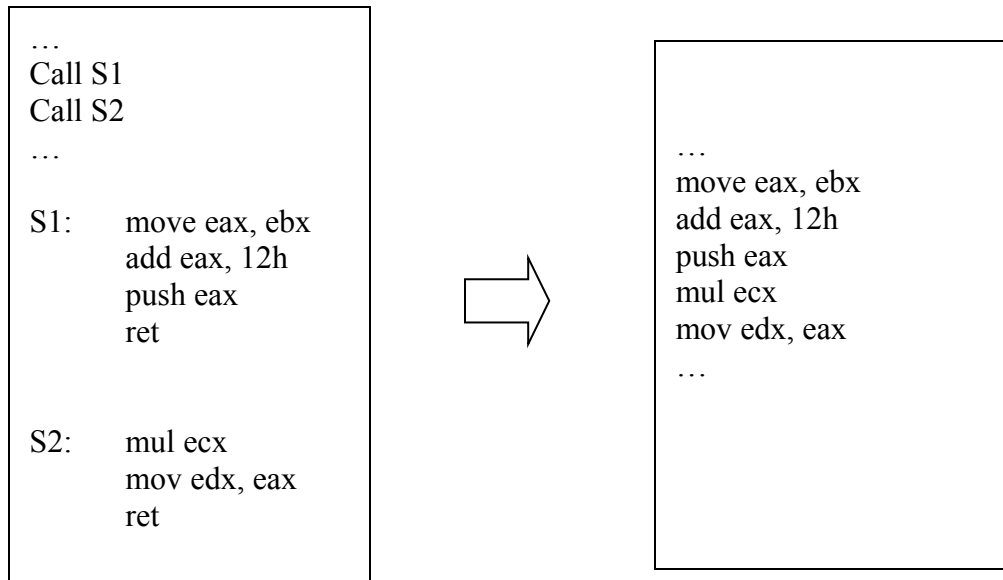


Figure 9: Subroutine Inlining

Figure 9 shows an example of subroutine inlining where call to subroutines S1 and S2 is replaced with its code.

Code outlining is reverse of code inlining. Code outlining converts a block of code into a subroutine and replaces the block with a call to the subroutine. This technique essentially does not preserve any logical code grouping [12].

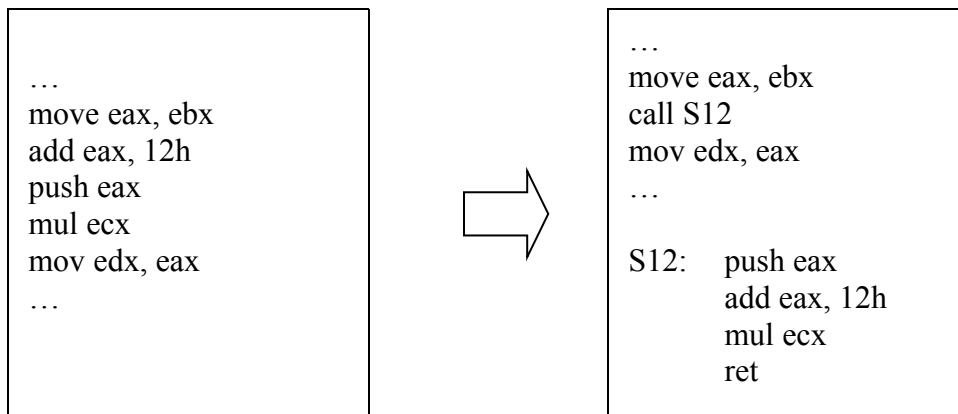


Figure 10: Subroutine Outlining

Figure 10 shows an example of subroutine outlining where subroutine S12 is created with randomly selected block of code.

6. Similarity Test

Metamorphic engine produces morphed copies of a single input program. Effective metamorphic engine will generate highly dissimilar copies. Similarity test is used to determine the diversity of the code generated by our metamorphic engine. We conducted repetitive similarity test to improve metamorphism of our engine. The similarity test compares two assembly programs and calculates the percentage of similarity between them. To compute the similarity between two files, we followed the following steps [11].

1. Given two assembly files a.asm and b.asm, extract opcode sequences from each file excluding comments, blank lines, labels, and other directives. Let's call these opcode sequences A and B for the files a.asm and b.asm respectively.
2. Consider m and n are the number of opcodes in A and B respectively.
3. Each opcode in A and B is assigned a number in ascending order i.e. first opcode is assigned 0, second opcode is assigned 1, third opcode is assigned 2, and so on.
4. Opcode sequences of A and B are divided into subsequences of length 3.
5. Every subsequence in A is compared with all subsequences in B. It is considered a match if the opcodes of any subsequence in A is same as the opcodes of any subsequence in B. These opcodes can be in any order. For example A is (mov,call,sub,add,test) and B is (mov,test,add,call,sub). The sequence (call,sub,add) in A matches with (add,call,sub) of B.
6. All such matches of A are computed and added together to find total number of match. This total number of matches is divided by m to get the similarity percentage of A (X).
7. Similarly the similarity percentage of B (Y) is computed.
8. The average of X and Y will give the actual similarity percentage between files a.asm and b.asm.

A graph is generated to visualize the similarity of the assembly files. Let's look at how a graph is generated:

1. Comparing two opcode sequences A and B, x axis represents opcode sequence A and y axis represents opcode sequence B.
2. A co-ordinate (12, 25) is marked if the subsequence (12, 13, 14) of A matches with the subsequence (25, 26, 27) of B.
3. A graph is generated by plotting all the matches for A and B (see figure 11-a).
4. But the graph in figure 11 is very populated. It is difficult to understand the similarity.
5. To generate a clean graph, all the matches less than some threshold are dropped. We assumed the threshold to be 5 and the graph in figure (11-a) is cleared in figure (11-b).

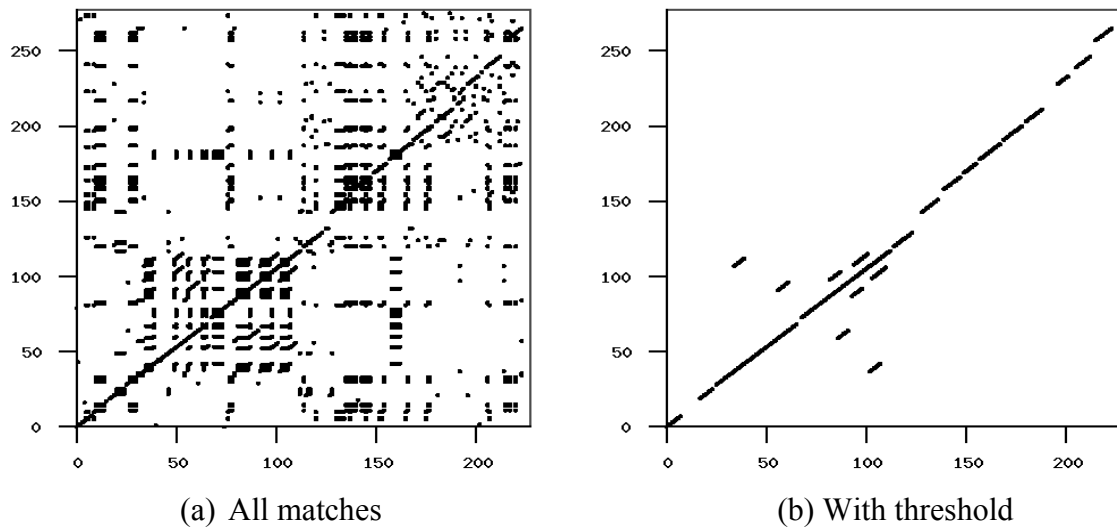


Figure 11: Similarity Graph

7. Hidden Markov Model

Hidden Markov Model also known as HMM is a statistical pattern analysis tool. HMM creates a model representing the input data. This input data is called training data. The training data consists of a list of unique symbols and their positional information in input sequence. HMM uses this model to determine if a given input sequence follows similar pattern as the model.

HMM is widely used for speech recognition and protein modeling. Recently HMM has been successfully used to detect metamorphic viruses [2, 9]. Metamorphic viruses are a family of viruses that changes in appearance while preserving the same functionality. Generally a family of viruses have similar pattern. Given a family of viruses HMM can come up with the statistical model representing the family. Now any virus can be tested against several such models to determine which family it belongs to.

Let's look at a simple example to understand inner working of HMM [14]. Suppose we want to determine annual temperatures of some distant location. The annual temperature can be either hot (H) or cold (C). We know the probability of a hot year followed by another hot year is 0.7 and a cold year followed by another cold year is 0.6. These probabilities are represented in matrix below,

$$\begin{array}{c}
 H \quad C \\
 \begin{array}{c} H \\ C \end{array} \left[\begin{array}{cc} 0.7 & 0.3 \\ 0.4 & 0.6 \end{array} \right]
 \end{array}$$

Figure 12: Temperature transition probability

We also know the correlation between tree sizes and temperature. Tree sizes are of three types small (S), medium (M), and large (L). The probability of tree being small in hot year is 0.1, medium is 0.4, and large is 0.5. Similarly the probability of tree being small in cold year is 0.7, medium is 0.2, and large is 0.1. The probabilistic relation between tree sizes and annual temperature is given by the matrix below,

$$\begin{array}{c} \\ H \\ C \end{array} \begin{array}{ccc} S & M & L \\ \left[\begin{array}{ccc} 0.1 & 0.4 & 0.5 \\ 0.7 & 0.2 & 0.1 \end{array} \right] \end{array}$$

Figure 13: Tree size probability

In this example, the annual temperatures are the states and the tree sizes are the observable symbols. The probability of different tree sizes at each temperature represents the probability of the observation symbols in each state. The states (H and C) are hidden since we can not see the temperature of distant location. We can only see the observation symbols (S, M, and L) which are statistically related to the states.

Suppose we have a sequence of observation symbols (S, M, S, L) of four consecutive years. We want to find out the sequence of states i.e. the annual temperature from the sequence of tree sizes.

The notations used in HMM:

T = Length of the observed sequence

N = Number of states in the model

M = number of distinct observation symbols

O = Observation sequence $\{O_0, O_1, \dots, O_{T-1}\}$

A = State transition probability matrix

B = Observation probability distribution matrix

π = Initial state distribution matrix

In this example, state transition probability matrix A, is the matrix with temperature transition probability (figure 12) with $N = 2$. The observation probability distribution matrix B, is the matrix of tree size probability (figure 13) with $M = 3$. Thus we get A and B as shown below,

$$A = \begin{bmatrix} 0.7 & 0.3 \\ 0.4 & 0.6 \end{bmatrix} \quad \text{and} \quad B = \begin{bmatrix} 0.1 & 0.4 & 0.5 \\ 0.7 & 0.2 & 0.1 \end{bmatrix}$$

The initial state distribution matrix, π represents the probability of being in a state initially. Consider the initial state distribution matrix for this example is

$$\pi = \begin{bmatrix} 0.6 & 0.4 \end{bmatrix}$$

The matrices A, B, and π forms the parameters of HMM model. Note that, the parameters A, B, and π are row stochastic, i.e. the summation of each row should be 1.

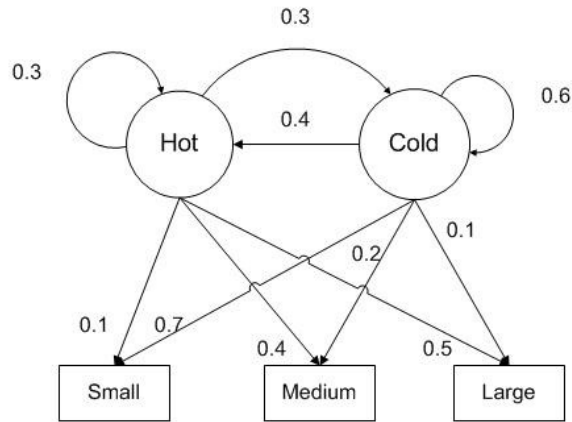


Figure 14: HMM Model

So far we have HMM model representing tree sizes and temperatures. Consider an observation sequence (S, M, S, L) of length $T = 4$. To determine the state transition for this sequence, HMM follows these steps:

1. Determine all possible state transitions = N^T .
2. Calculate the probability of given observation sequence for each state transition of step 1. The formula used to calculate this probability is:

$$\begin{aligned}
 P(\text{HHCC}) &= \pi_H * b_H(\text{S}) * a_{H,H} * b_H(\text{M}) * a_{H,C} * b_C(\text{S}) * a_{C,C} * b_C(\text{L}) \\
 &= (0.6) * (0.1) * (0.7) * (0.4) * (0.3) * (0.7) * (0.6) * (0.1) \\
 &= 0.000212
 \end{aligned}$$

Table 3 shows list probabilities of observing (S, M, S, L) for all possible state sequences.

3. The state sequence with highest probability is selected. The state sequence "CCCH" has the highest probability in this example.

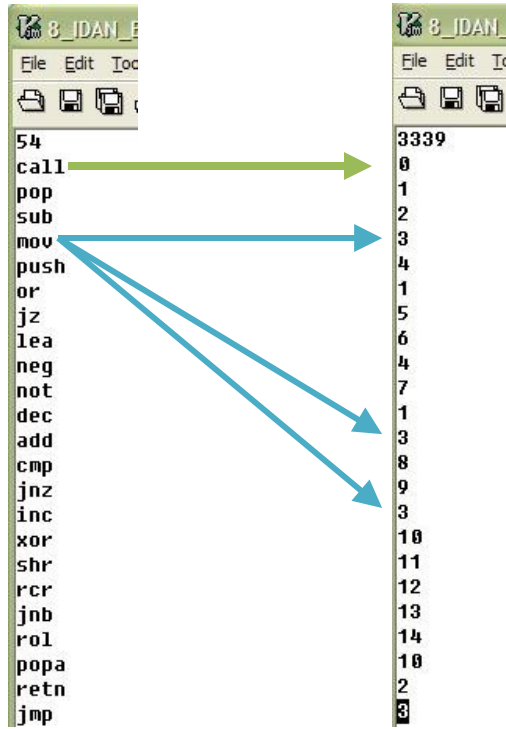
Table 3: Probabilities of observing (S, M, S, L) for all possible state sequences

state sequence	probability
<i>HHHH</i>	0.000412
<i>HHHC</i>	0.000035
<i>HHCH</i>	0.000706
<i>HHCC</i>	0.000212
<i>HCHH</i>	0.000050
<i>HCHC</i>	0.000004
<i>HCCH</i>	0.000302
<i>HCCC</i>	0.000091
<i>CHHH</i>	0.001098
<i>CHHC</i>	0.000094
<i>CHCH</i>	0.001882
<i>CHCC</i>	0.000564
<i>CCHH</i>	0.000470
<i>CCHC</i>	0.000040
<i>CCCH</i>	0.002822
<i>CCCC</i>	0.000847
Σ probability	0.009629
max probability	0.002822

Therefore the most probable state sequence for given observation sequence is CCCH.

7.1 HMM as Virus Detection Tool

HMM as virus detection tool requires training data to produce a model. The training data consists of observation sequence and unique symbols. The observation sequence and unique symbols are derived from several viruses of a family. These viruses are programs written in assembly language. The observation symbols are unique assembly opcodes among all viruses. The opcodes of all viruses are concatenated to produce one long observation sequence. HMM is trained on this observation sequence to produce the model. An example of such observation sequence is shown in figure 15. The model is shown in figure 16.



(a) Unique Symbols

(b) Observation sequence

Figure 15: Training Data

8_IDAN_N3_E4.model (-My Documents\CSC297\Source\HMMModel) - GVIM

File Edit Tools Syntax Buffers Window Help

N=3, M=54, T=3339

I:

1.0000000000000000	0.0000000000000000	0.0000000000000000
--------------------	--------------------	--------------------

A:

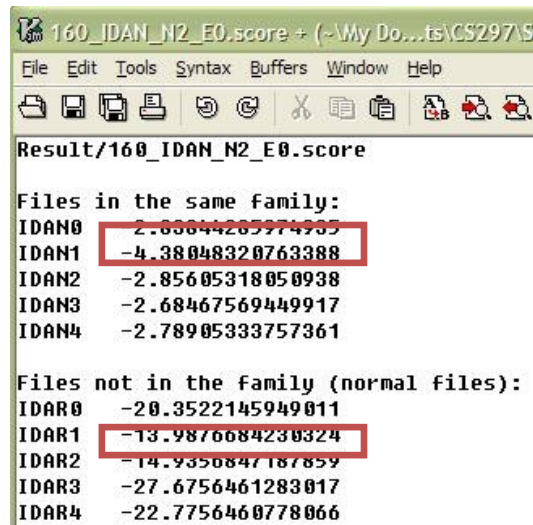
0.75086317967996	0.03798198915413	0.21115483116590
0.09567091277231	0.83038682159473	0.07394226563295
0.10271478146879	0.07750068809967	0.81978453043154

B:

call	0.16029021641704	0.03254006951896	0.01716054612392
pop	0.11337062907665	0.0000000000000000	0.04133515477480
sub	0.00422611246341	0.07267169820262	0.06590174434185
mov	0.03062452304487	0.08965929951014	0.43057414360768
push	0.34869944435288	0.0000000000000000	0.03446530803580
or	0.0000000000000000	0.01138701204944	0.01198391194381
jz	0.0000000000000000	0.13995532443008	0.0000000000000000
lea	0.01463341373056	0.00152189875803	0.01857485328313
neg	0.00221808307166	0.01269378392105	0.00103496710605
not	0.00170165838178	0.01138959044484	0.00080582051881
dec	0.00166177057784	0.04686500279155	0.01045328354292

Figure 16: HMM model

Given a virus to test against HMM model, HMM produces following result file:



```
160_IDAN_N2_E0.score + (~\My Do...ts\CS297\A
File Edit Tools Syntax Buffers Window Help
Result/160_IDAN_N2_E0.score
Files in the same family:
IDAN0 2.63644265774735
IDAN1 -4.38048320763388
IDAN2 -2.85605318050938
IDAN3 -2.68467569449917
IDAN4 -2.78905333757361
Files not in the family (normal files):
IDAR0 -20.3522145949011
IDAR1 -13.9876684230324
IDAR2 -14.9350847187059
IDAR3 -27.6756461283017
IDAR4 -22.7756460778066
```

Figure 17: The Result File

In the result file, IDAN0 to IDAN4 are the viruses from the same family. The score for these viruses is greater than -4.38 which are defined as a threshold. A file with a score less than the threshold is not considered as part of this family. The files IDAR0 to IDAR4 have scores less than the threshold and therefore not in the family.

8. Implementation

8.1 Introduction

In general metamorphic engine has to implement some or all code obfuscation techniques. In addition to using these techniques, each implementation will have its own heuristics. These heuristics may include processes that decide type of obfuscation techniques to use, when to apply them, and how to apply them.

We started our implementation by following some of the existing metamorphic engines like Evol. Evol is a metamorphic virus that used code obfuscation techniques such as dead code insertion, register / operands usage exchange, and equivalent instruction substitution. In addition to the techniques used by Evol, we added few more variations of these techniques. This section gives detailed explanation of the code obfuscation techniques we used.

8.2 Goals

Our implementation was geared toward achieving following goals:

- Generate morphed copies of a single input virus. These morphed copies should have minimum similarity with the base virus and among themselves.
- The morphed copies should have same functionality as the base virus.
- Morphed copy should be close to normal program. Assumption here is the normal programs are the cygwin utility files of the same size as the base virus. The reason behind using cygwin utility files is they probably are doing same low level operations as a virus.
- The metamorphic engine should work on any assembly program.

8.3 Code Obfuscation Techniques Used

8.3.1 Dead Code Insertion

Dead code insertion is adding NOP or do-noting instructions. We used dead code insertion to introduce opcodes that are alien to the base virus. The alien opcodes were determined by analyzing the base virus and normal programs.

We first generated statistics of the base virus to find out all the opcodes used. The graph in figure 18 below lists the opcodes used in the base virus with their frequency.

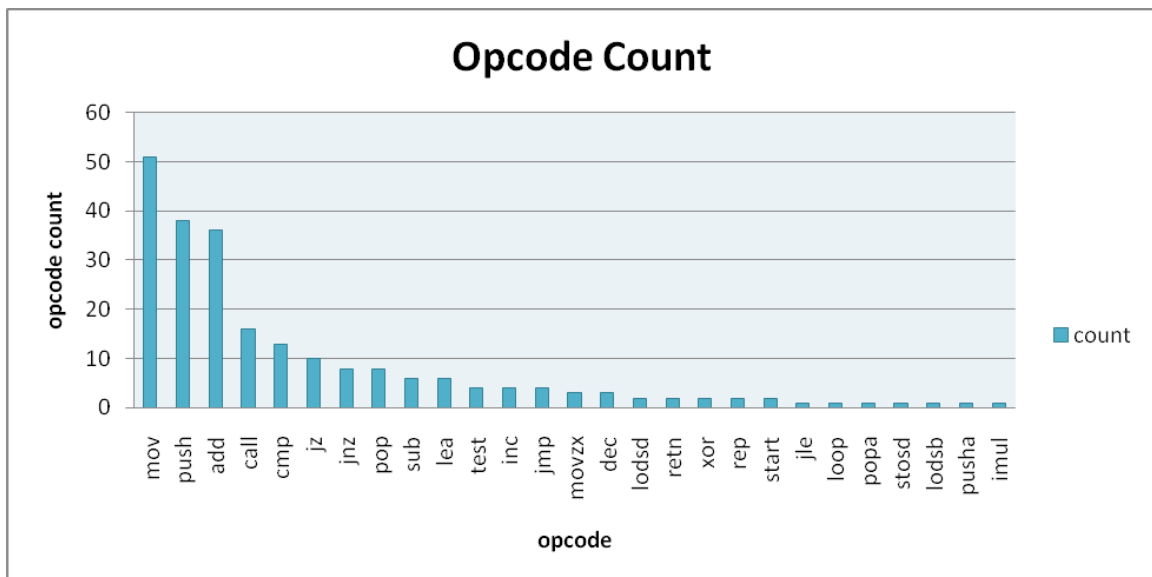


Figure 18: Base virus opcodes and their frequency

Our base virus has 27 unique opcodes and six of them appear more than 10 times. Opcodes mov, push, add, call, cmp, and jz are the most frequent appearing opcodes. We designed our dead opcode set to include more of the infrequent used opcodes.

We then analyzed the normal program for its opcode frequency. The graph in figure 19 shows the statistics of a normal file.

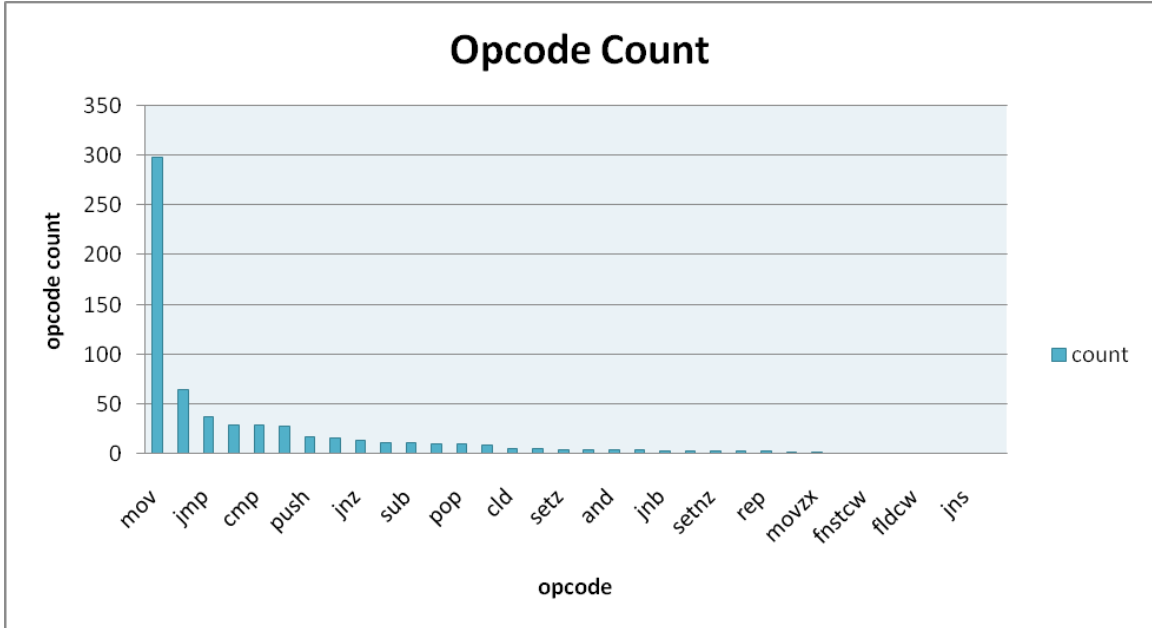


Figure 19: Opcodes of normal file and their frequency

When the statistics of a normal file is compared with the base virus, we get the list of opcodes that are unique to a normal file. The unique opcodes are *AND, INT, FNSTCW, OR, FLDCW, LEAVE, JNS, SETNZ, SETZ, JB, CLD, JNB, SHL, INC, FLD, FSTP, and REPE*.

This comparison shows that the above unique opcodes should be included in morphed copies to make them look more like a normal file. Based on this conclusion the dead code instructions are modeled to include most of the above unique opcodes. The table 4 shows some examples of dead code instructions used. Refer to Appendix A for complete list of dead code instruction.


Table 4: Arithmetic Dead Code Instructions

1. add R, 0
2. sub R, 0
3. adc bx, 0
4. sbb bx, 0
5. inc R followed by dec R

These dead code instructions are injected at randomly selected locations in the base virus. For every selected location, we insert a single dead code instruction. The dead code

instruction to be inserted is randomly selected. These are categorized as simple single NOP instruction substitution.

As the variation to simple single NOP instruction substitution, we introduced unconditional jump NOP instruction substitution. The jump NOP works by introducing unconditional jump to next immediate instruction. An example of this variation is shown below.

`mov edx, [esi+entryPoint]`  `pl010235: jmp pl010235`
`pl010235: mov edx, [esi+entryPoint]`

8.3.1.1 NOP sequence insertion

Dead code insertion was used to insert a single NOP Instruction. In NOP sequence insertion, a random sequence of NOP instructions are inserted at randomly selected locations. The locations to insert NOP sequence were categorized in two viz. beginning of the code section and rest of the code section. To insert or not to insert a NOP sequence in the beginning of the code section is decided randomly. While for the rest of the code section, the insertion location and a NOP sequence is selected randomly.

1. Determine entry point of a virus.
2. Generate random number between 0 to 3
3. If the random number is 0 then insert NOP sequence
4. To inset NOP sequence:
 - a. Randomly select length of a NOP sequence from 3, 5, and 7.
 - b. Generate random permutation of the above selected length. For example if the length selected is 3 then 2^3 permutations are possible, randomly select any sequence out of 8 permutations.
 - c. Insert this sequence into a virus.

Figure 20: Algorithm to insert NOP sequence on entry point

1. Generate random number between 0 to 50
2. Add a constant number to get value X
3. For every X instruction in the base virus insert randomly selected NOP sequence.

Figure 21: Algorithm to insert random NOP sequence

8.3.1.2 Transformations of Evol

Along with a single dead code insertion and a NOP sequence insertion, we introduced some new dead code insertions. These insertions are inspired from Evol virus [6]. Evol virus substitutes a single instruction by surrounding it with dead code. The Evol transformations used here are listed in table 5.

Table 5: Evol transformations [6]

Reg - Register (i.e. EAX, EBX)
 Mem - Memory address (i.e. [EAX])
 r/m - Register or Memory
 imm - Immediate Value (i.e. OP Reg, ACABh)

OP = {ADC, ADD, AND, CMP, OR, SBB, SUB, XOR}
 OP1 = {DIV, IDIV, IMUL, MUL, NEG, NOT, TEST}
 OP2 = {RCL, RCR, ROL, ROR, SAL, SAR, SHL, SHR}

Original	Transformed
- MOV r/m, reg - MOV reg, r/m - TEST r/m, reg - LEA r32, mem - OP r/m, reg - OP reg, r/m	PUSH RandomReg MOV RandomReg, OriginalReg ADD RadnomReg, RandomImm8 OP r/m - RandomReg, OriginalReg POP RandomReg
- MOV r/m, reg - TEST r/m, reg - OP r/m, reg	PUSH RandomReg MOV RandomReg, OriginalReg OP OriginalR/M, RandomReg POP RandomReg
- MOV reg, r/m - LEA reg, mem - OP reg, r/m	PUSH RandomReg MOV RandomReg, OriginalReg OP RandomReg, OriginalR/M MOV OriginalReg, RandomReg POP RandomReg
- OP r/m8, imm8 - MOV r/m8, imm8 - TEST r/m8	PUSH RandomReg MOV RandomReg8, Imm8 OP OriginalR/M8, RandomReg8 POP RandomReg

One disadvantage with these transformations is an instruction is substituted with a block of instructions beginning with push followed by some instructions and ending with pop. Therefore these transformations increase the number of push and pop opcodes. This also creates a pattern of starting with push and ending in pop [20].

8.3.2 Equivalent instruction substitution

Some opcodes appear frequently in the base virus like mov, push, add, call, cmp, and jz. To minimize the number of these opcodes, we used equivalent instruction substitution. In an equivalent instruction substitution, an instruction is replaced with another instruction or a block of instructions with the same functionality. For example substitutions for add are listed in table 6.

Table 6: Substitutions for add

add R, imm	1. sub R, new_imm where new_imm = imm x (- 1) 2. lea R, [R + imm]
add R, 1	1. not R neg R

Here, opcode add is replaced with opcodes like “sub”, “lea”, and “not” followed by “neg”. Similarly opcodes like mov, cmp, test etc are replaced with equivalent instructions. The complete list can be found in appendix B.

The substitution for each instruction is decided based on the type of operands like

<i>REG (8), REG (8)</i>	<i>REG (16), REG (16)</i>	<i>REG (32), REG (32)</i>
<i>REG (8), MEM</i>	<i>REG (16), MEM</i>	<i>REG (32), MEM</i>
<i>REG (8), IMM</i>	<i>REG (16), IMM</i>	<i>REG (32), IMM</i>
<i>MEM, REG (8)</i>	<i>MEM, REG (16)</i>	<i>MEM, REG (32)</i>
<i>MEM, IMM</i>		

8.3.3 Transpose

After a morph copy is generated using dead code insertion and equivalent substitution, we apply transpose to generate final output.

1. Read two instructions with 2 operands.
2. Generate a random number between 0 and 3.
3. If the random number is 0 then perform transpose.
4. To perform transpose:
 - a. Read third instruction.
 - b. If the third instruction is not any conditional jump instruction then
 - i. If to-operands of both instructions are not equal
and
to-operand of first instruction is not equal to from-operand of second instruction
and
from-operand of first instruction is not equal to to-operand of second instruction
1. Swap two instructions.

Figure 22: Algorithm for transpose

The basic transpose algorithm applies only to instructions with register operands. We extended this algorithm to include instructions with memory operands. To achieve this extension, we added a new condition check. While comparing the operands in both the instructions, we had to make sure that none of the registers are used as memory pointers. For example following two instructions can be swapped.

```
mov  ax, cx
add  [dx + 2], 5
```

The following two instructions can not be swapped.

```
mov  ax, cx
add  [ax + 2], 5
```

The high level algorithm of our metamorphic engine is shown in figure 23.

1. Determine the start of code section.
2. $RAND_NUM = \text{random number between } 0 \text{ and } 3$.
3. If $RAND_NUM = 0$ then perform *NOP sequence insertion at entry point*.
4. $RAND_NUM = \text{random number between } 50 \text{ and } 100$
5. For every $RAND_NUM$ instruction, perform *random NOP sequence insertion*.
6. $RAND_NUM_SUB = \text{random number between } 0 \text{ and } 3$
7. If $RAND_NUM_SUB = 0$ then select the instruction for Substitution // substitution is done for about 1 in 4 instructions.
8. Substitution:
 - a. $RAND_DEAD_EQUI = \text{random number between } 0 \text{ and } 3$.
 - b. If $(RAND_DEAD_EQUI < 2)$
//equivalent code substitution is done 66%
 - i. Perform *equivalent code substitution*
 - c. Else
 - i. Perform *dead code insertion*
//randomly select among Single NOP instruction insertion, // jump NOP, and Evol transformations.
9. Repeat steps 5 to 8 till end of the file.
10. Perform *transpose* on the generated morphed code.

Figure 23: High level algorithm of Metamorphic Engine

9. Experiments

We generated a large number of metamorphic virus variants of the base virus with our metamorphic engine. The metamorphic virus variants were generated by applying the metamorphic engine iteratively over a single base virus. Applying the metamorphic engine once on an input is 1st generation metamorphism. Applying the metamorphic engine twice on an input is 2nd generation metamorphism and so on.

The metamorphic engine can take any assembly program as input. The output is a morphed copy of the input. These assembly sources are then compiled into executables using FASM [21]. These executables are then disassembled using IDA Pro with default settings (686 instruction set) [22]. These assembly programs were used to perform all tests. To keep the tests more realistic IDA-pro generated assembly files were used rather than the original assembly source from the engine.

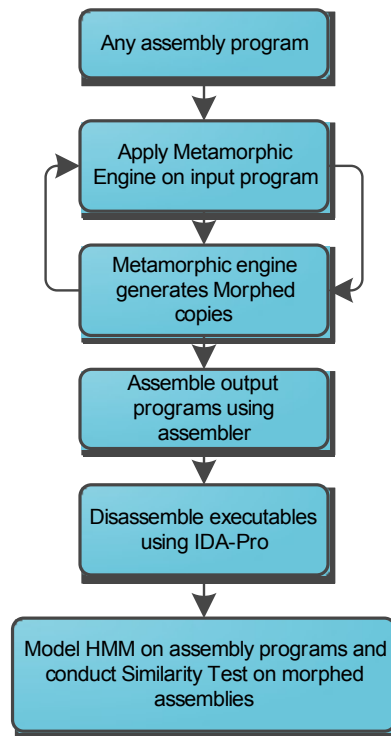


Figure 24: Over all Process

All our tests were performed using two different tools. These include Commercial virus scanner, Similarity Test, and statistical pattern analysis tool such as Hidden Markov Model.

9.1 Commercial virus scanner

In our testing, the base virus was successfully detected and quarantined by the commercial virus scanner installed on our machine. But the same virus scanner failed to detect morphed copies of the base virus.

9.2 Similarity Test

Similarity test compares and reports the percentage of similarity of two assembly programs. The purpose of the similarity test is to measure the code diversity of the morphed copies.

We compared the base virus with 1st to 9th generations of metamorphic copies. These comparisons were performed using the default settings of similarity test i.e. 10 opcodes in a sequence is considered a match. The result of this test is shown below in figure 25. The similarity between the base virus and 1st generation virus is about 70%. The similarity

decreases with higher generations. 9th generation virus is about 10% similar to the base virus.

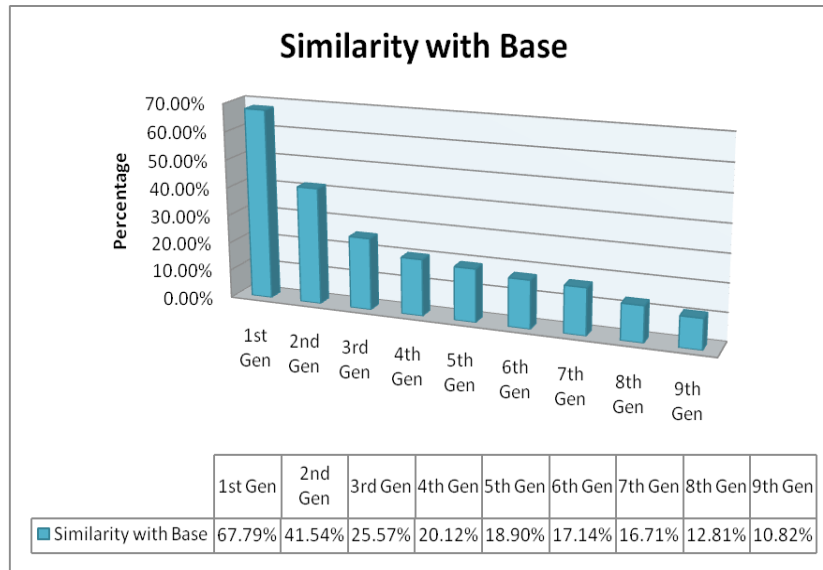


Figure 25: Similarity results of the base virus v/s 9 different generations

After applying the metamorphic engine to the base virus, the number of opcodes in morphed copies increases. The dissimilar length of the compared files may affect similarity test. So we compared a pair of viruses from the same generation. The viruses from the same generation are of similar length. 1st generation viruses are about 50% similar whereas 9th generation viruses are about 2.5% similar as shown in figure 26. Note that, the viruses generated by Next Generation Virus Creation Kit (NGVCK) were found to be about 10% similar with default settings [2]. Based on these similarity tests, we decided to model HMM on highly dissimilar generation which is 9th generation.

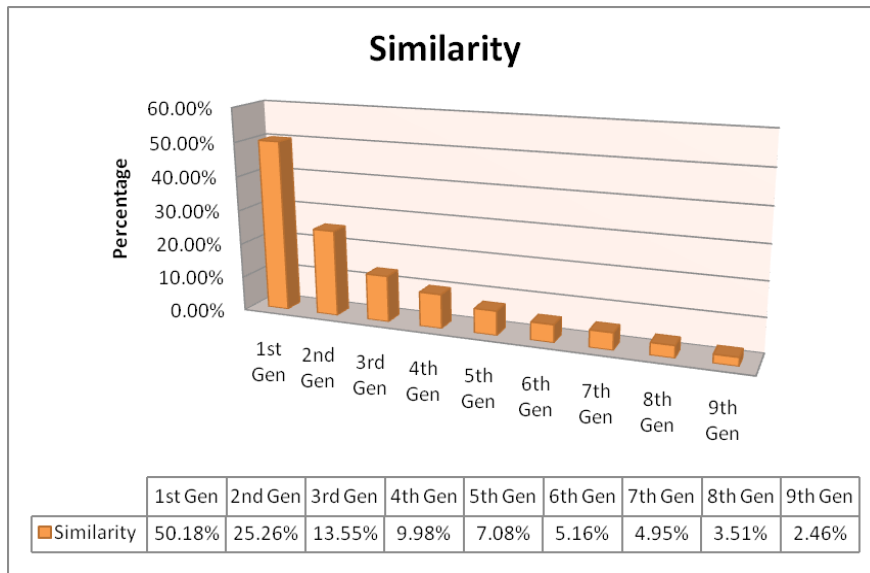


Figure 26: Graph of Similarity of two N generations

9.3 HMM

Similarity test shows that 9th generation viruses are highly metamorphic. To further test morphed copies, the statistical pattern analysis tool such as HMM was used. This test consists of four test cases:

1. N generation viruses against the base virus model
2. The base virus against the morphed virus model
3. Normal files against 9th generation virus model
4. Morphed viruses against normal file model

The idea of this test is to compare statistics of morphed copies with the base virus and normal files.

9.3.1 N generation viruses against the base virus model

We trained HMM on 60 copies of the base virus with $N = 2$ and compared 9 different generations of viruses against this model. The base virus model is listed in appendix D. The 1st generation virus scored about -69 and next generations are showing low scores. The statistical pattern of N different generations is different than the base virus.

Table 7: HMM of base virus tested with 9 generations

Virus	Score
1 st Generation	-68.722928938174
2 nd Generation	-131.862876167904

3 rd Generation	-198.857278862957
4 th Generation	-234.377340367938
5 th Generation	-261.32928056904
6 th Generation	-297.823863014344
7 th Generation	-319.359839713903
8 th Generation	-338.517130927289
9 th Generation	-343.070315142923

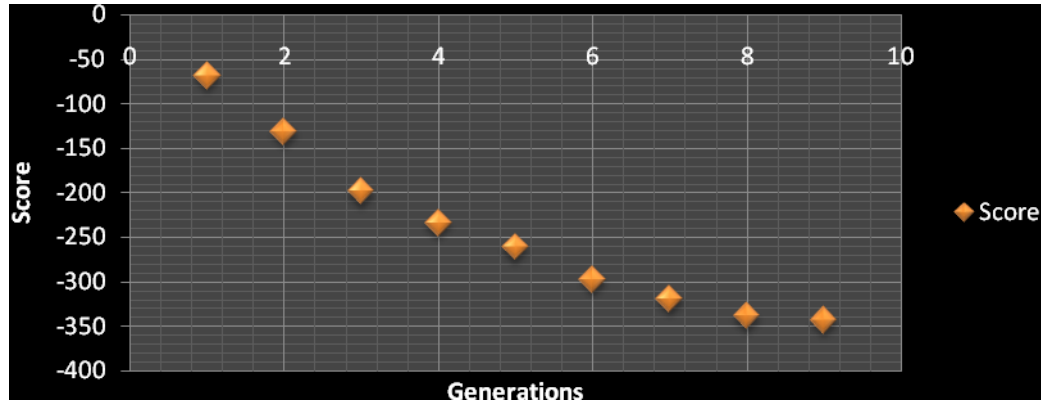


Figure 27: N (1-9) generation viruses tested against base virus model

9.3.2 The Base virus against the morphed virus model

We then modeled HMM for odd generations of viruses. The base virus was tested against these modes and scores are listed in table 8. Results shows the statistical pattern of the base virus can still be detected by different generation of viruses.

Table 8: The base virus tested against N Generation Model

Model	Score
1 st Generation Model	-2.26519095918038
3 rd Generation Model	-2.5616088296304
5 th Generation Model	-2.7804691006756
7 th Generation Model	-6.53547571903687
9 th Generation Model	-9.36420192759975

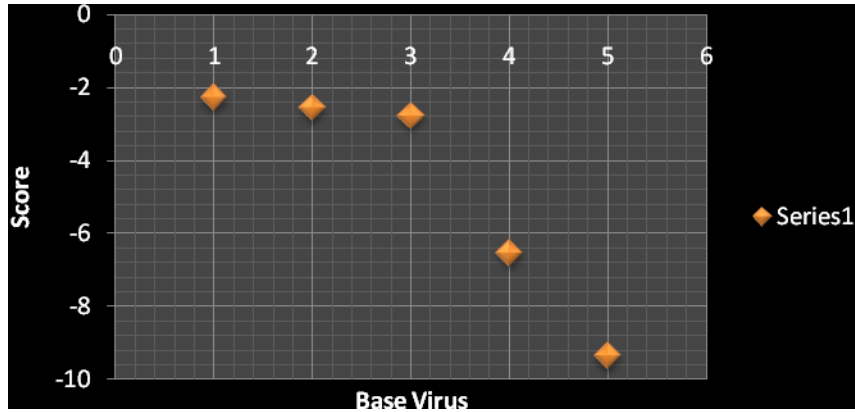


Figure 28: Base virus tested against N generation models

9.3.3 Normal files against 9th generation virus model

We collected 120 viruses from 9th generation and generated HMM model of that family. We used 4 fold cross validation i.e. HMM was modeled on 90 viruses and 30 viruses tested against this model. The model was generated with 2 states. The threshold for the family is -4.2650. Any file scoring higher than the threshold is considered to be family virus and a file having score less than threshold is considered a non-family file. Normal files were tested against this model. Out of 30 normal files, the maximum score -11.7943 is less than the threshold. So all normal files are identified correctly and declared non-family files. This gives 0% false positives and 0% false negatives.

Table 9: Results of 9th generation viruses tested against 9th generation model

9 th Generation Model with N =2							
Family Viruses				Normal Files			
G9_0	-3.1677	G9_15	-4.2650	N0	-14.4239	N15	-356.9657
G9_1	-3.1684	G9_16	-3.1277	N1	-42.9527	N16	-34.4798
G9_2	-3.1269	G9_17	-3.1266	N2	-444.9695	N17	-11.7943
G9_3	-3.1419	G9_18	-3.1248	N3	-532.4239	N18	-406.5270
G9_4	-3.1596	G9_19	-3.1138	N4	-20.8160	N19	-406.5270
G9_5	-3.1692	G9_20	-3.1250	N5	-18.7624	N20	-507.2849
G9_6	-3.1419	G9_21	-3.1486	N6	-20.8160	N21	-15.2849
G9_7	-3.1782	G9_22	-3.1517	N7	-17.2520	N22	-507.2849
G9_8	-3.1115	G9_23	-3.1661	N8	-27.8287	N23	-473.7664
G9_9	-3.1305	G9_24	-3.1420	N9	-19.0357	N24	-356.7943
G9_10	-3.1404	G9_25	-3.1743	N10	-406.5270	N25	-36.2016
G9_11	-3.1262	G9_26	-3.1522	N11	-37.8043	N26	-32.1237
G9_12	-3.1299	G9_27	-3.1638	N12	-25.4653	N27	-507.2849
G9_13	-3.1424	G9_28	-3.2038	N13	-23.9582	N28	-35.0315
G9_14	-3.1300	G9_29	-3.1714	N14	-25.2204	N29	-356.9657
Min Score = -4.2650				Max Score = -11.7943			

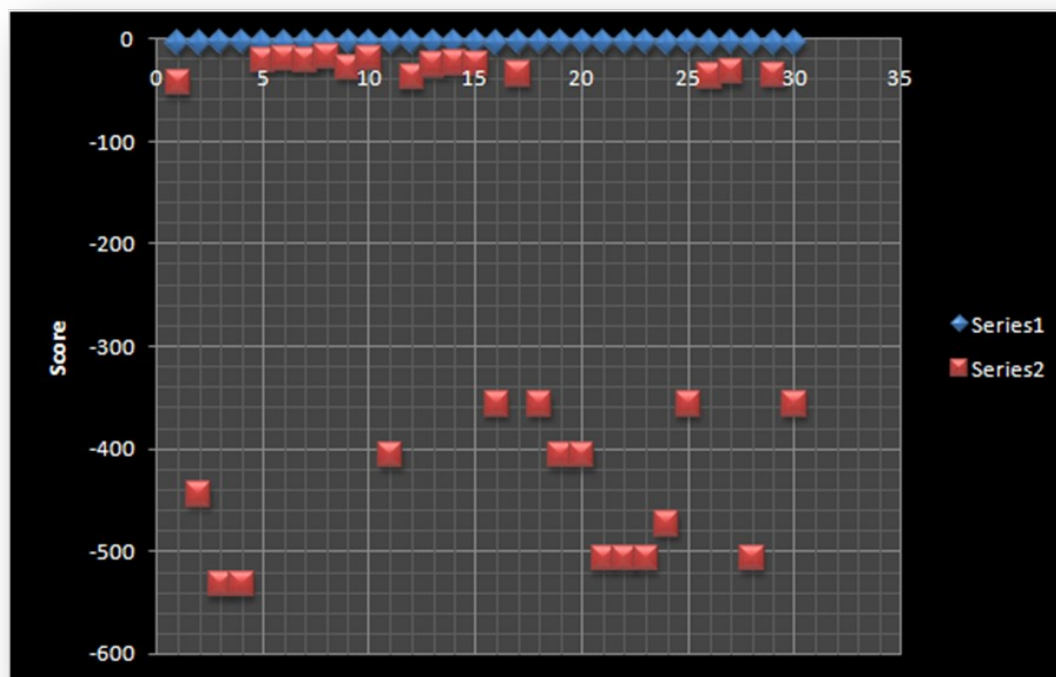


Figure 29: Family viruses and normal files tested against 9th generation model

9.3.4 Morphed viruses against normal file model

We collected 40 cygwin files as a set of normal files. We generated HMM model on a set of normal files. Then 9th generation viruses are tested against this model. The threshold for normal files is -180.5254. All 9th generation viruses scored higher than the threshold. The maximum score of 9th generation viruses is -37.2978. So the 9th generation viruses are considered as normal files. This is 100% false positives.

Table 10: Results of 9th generation viruses tested against normal model

Normal model with N = 2			
Normal Files		9 th Generation Viruses	
N0	-21.9658	G9_0	-173.3586
N1	-5.20571	G9_1	-160.9587
N2	-180.5254	G9_2	-154.1496
N3	-4.53708	G9_3	-159.1445
N4	-1.7961	G9_4	-168.9089
N5	-1.7246	G9_5	-169.4739
N6	-1.7961	G9_6	-164.7176
N7	-2.0771	G9_7	-37.2978
N8	-2.0542	G9_8	-169.2335
N9	-1.7599	G9_9	-158.5317
Min Score = -180.5254		Max Score = -37.2978	

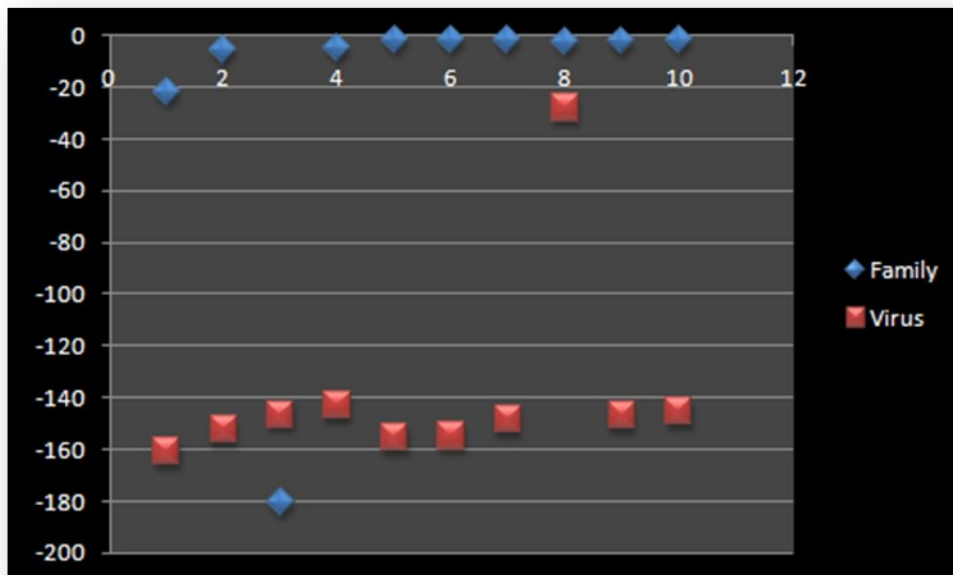


Figure 30: Family viruses and 9th generation viruses tested against normal model

HMM model of normal files has very low threshold. The reason behind this low threshold is less similarity within a set of normal files. With less similarity, generating most probable model is difficult. And this is causing false positives.

10. Conclusion

We developed the metamorphic engine producing morphed copies of the base virus that are highly dissimilar and includes some opcodes of the normal program. These were the two main criteria described in [2] which are required in metamorphic virus to defeat HMM. In our engine, we employed code obfuscation techniques such as equivalent instruction substitution, dead code insertion, and transpose. We introduced floating point opcodes in morphed copies which are commonly found in normal programs.

The similarity showed that the morphed copies are highly metamorphic with 2.5% similarity index. Even with such a high metamorphism, HMM was able to classify the morphed copies of the base virus as the family virus. The base virus was compared with model of morphed copies, HMM was still able to classify the base virus as the same family. This fact proves that even with high metamorphism, HMM is able to identify a common statistical pattern across all morphed copies and the base virus. HMM has proved very difficult to defeat.

11. Future Work

We implemented code obfuscation techniques such as equivalent instruction substitution, dead code insertion, and transposition. The next step would be to include more code obfuscation techniques into a metamorphic engine. Also, applying different subset of code obfuscation techniques can generate more diverse morphed copies.

The size of the base virus is 1.5KB. Applying our metamorphic engine iteratively changes the original file size. 1st generation morphed files are about 2 KB which 35 % more than the original size. The graph in figure 31 reflects the increase in file size over generations. A technique can be devised to implement a metamorphic engine such that file sizes of the morphed copies do not change.



Figure 31: Change in file sizes over 9 generations.

One of the techniques to make viruses look like normal programs is to compare the HMM model parameters of a virus and normal files. The matrix B shows the probabilities of observation symbols in all states. This matrix can be converted to a state transition table. The state transition tables of virus and normal programs can be compared to change the statistics of a virus. This may make virus look alike normal program.

REFERENCES

- [1] M. Stamp, "Information Security: Principles and Practice," August 2005.
- [2] W. Wong, "Analysis and Detection of Metamorphic Computer Viruses," Master's thesis, San Jose State University, 2006.
<http://www.cs.sjsu.edu/faculty/stamp/students/Report.pdf>
- [3] S. Attaluri, "Profile hidden Markov models for metamorphic virus analysis," Master's thesis, San Jose State University, 2007.
http://www.cs.sjsu.edu/faculty/stamp/students/Srilatha_cs298Report.pdf
- [4] P. Szor, "The Art of Computer Virus Defense and Research," Symantec Press 2005.
- [5] VX Heavens, <http://vx.netlux.org/>
- [6] Orr, "The viral Darwinism of W32.Evol: An in-depth analysis of a metamorphic engine," 2006. <http://www.antilife.org/files/Evol.pdf>
- [7] Orr, "The molecular virology of Lexotan32: Metamorphism illustrated," 2007. <http://www.antilife.org/files/Lexo32.pdf>
- [8] E. Konstantinou, "Metamorphic Virus: Analysis and Detection," January 2008.
- [9] A. Venkatesan, "Code Obfuscation and Metamorphic Virus Detection," Master's thesis, San Jose State University, 2008.
http://www.cs.sjsu.edu/faculty/stamp/students/ashwini_venkatesan_cs298report.doc
- [10] The Mental Driller, "Metamorphism in practice or How I made MetaPHOR and what I've learnt," February 2002. <http://vx.netlux.org/lib/vmd01.html>
- [11] P. Mishra, "A taxonomy of software uniqueness transformations," December 2003.
<http://www.cs.sjsu.edu/faculty/stamp/students/FinalReport.doc>
- [12] J. Aycock, "Computer Viruses and malware," Springer Science+Business Media, 2006.
- [13] E. Daoud and I. Jebril, "Computer Virus Strategies and Detection Methods," Int. J. Open Problems Compt. Math., Vol. 1, No. 2, September 2008.
[http://www.emis.de/journals/IJOPCM/files/IJOPCM\(vol.1.2.3.S.08\).pdf](http://www.emis.de/journals/IJOPCM/files/IJOPCM(vol.1.2.3.S.08).pdf)
- [14] M. Stamp, "*A Revealing Introduction to Hidden Markov Models*," January 2004.
<http://www.cs.sjsu.edu/faculty/stamp/RUA/HMM.pdf>

- [15] Walenstein, R. Mathur, M. Chouchane R. Chouchane, and A. Lakhotia, "The design space of metamorphic malware," In Proceedings of the 2nd International Conference on Information Warfare, March 2007.
- [16] R. Grimes. Malicious Mobile Code: Virus Protection for Windows. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2001.
- [17] F. Cohen, "Computer viruses: theory and experiments," Computer Security, 6(1):22-35, 1987.
- [18] "Benny/29A", Theme: metamorphism,
<http://www.vx.netlux.org/lib/static/vdat/epmetam2.htm>
- [19] J. Borello and L. Me, "Code Obfuscation Techniques for Metamorphic Viruses", Feb 2008, <http://www.springerlink.com/content/233883w3r2652537>
- [20] A. Lakhotia, "Are metamorphic viruses really invincible?" Virus Bulletin, December 2005.
- [21] FASM, <http://flatassembler.net/>
- [22] IDA Pro, <http://www.hex-rays.com/idapro/>

Appendix A: Dead code instructions

Transfer Dead Code

1. `mov R, R`
2. `push R` followed by `pop R`

Arithmetic Dead Code

1. `add R, 0`
2. `sub R, 0`
3. `adc bx, 0`
4. `sbb bx, 0`
5. `inc R` followed by `dec R`

Logical Dead Code

1. `shl R, 0`
2. `shr R, 0`
3. `and R, 1`
4. `test R, 1`
5. `or R, 0`
6. `xor R, 0`

Floating Point Dead Code

1. `fadd st2, st0`
2. `fmul st2, st0`
3. `fld st2`
4. `fsub st2, st0`
5. `fdiv st2, st0`
6. `fst st3`

Miscellaneous Dead Code

1. `nop`
2. `neg R, not R, dec R`

Appendix B: Equivalent instruction substitution

Notations:

R – Register (eax, ax, ah, al)

RR – Random register

mem, [mem] – Memory address ([esi])

imm – Immediate value (12h)

op1 – To-operand with length more than 1 including R and mem

op2 – From-operand with length more than 1 including R, mem, and imm

loc – any location or label

add R, imm	3. sub R, new_imm where new_imm = imm x (- 1) 4. lea R, [R + imm]
add R, 1	3. not R neg R
mov R, imm	1. mov R, random_imm add R, new_imm where new_imm = imm - random_imm 2. mov R, random_imm sub R, new_imm where new_imm = (random_imm - imm) mov R, random_imm xor R, new_imm
mov R1, R2 (no 8 bit R)	1. push R2 pop R1
mov R, mem (no 8 bit R)	1. push mem pop R
mov R, imm (no 8 bit R)	1. push imm pop R 2. lea R, [imm]
mov mem, R (no 8 bit R)	1. push R pop mem
mov mem, imm	1. push imm pop mem
cmp R, 0	1. or R, R 2. and R, R 3. test R, R
cmp R1, R2	1. sub R1, R2
cmp R, mem	1. sub R, mem
cmp R, imm	1. sub R, imm
cmp mem, R	1. sub mem, R
cmp mem, imm	1. sub mem, imm
and R1, R2	1. push RR mov R, R1

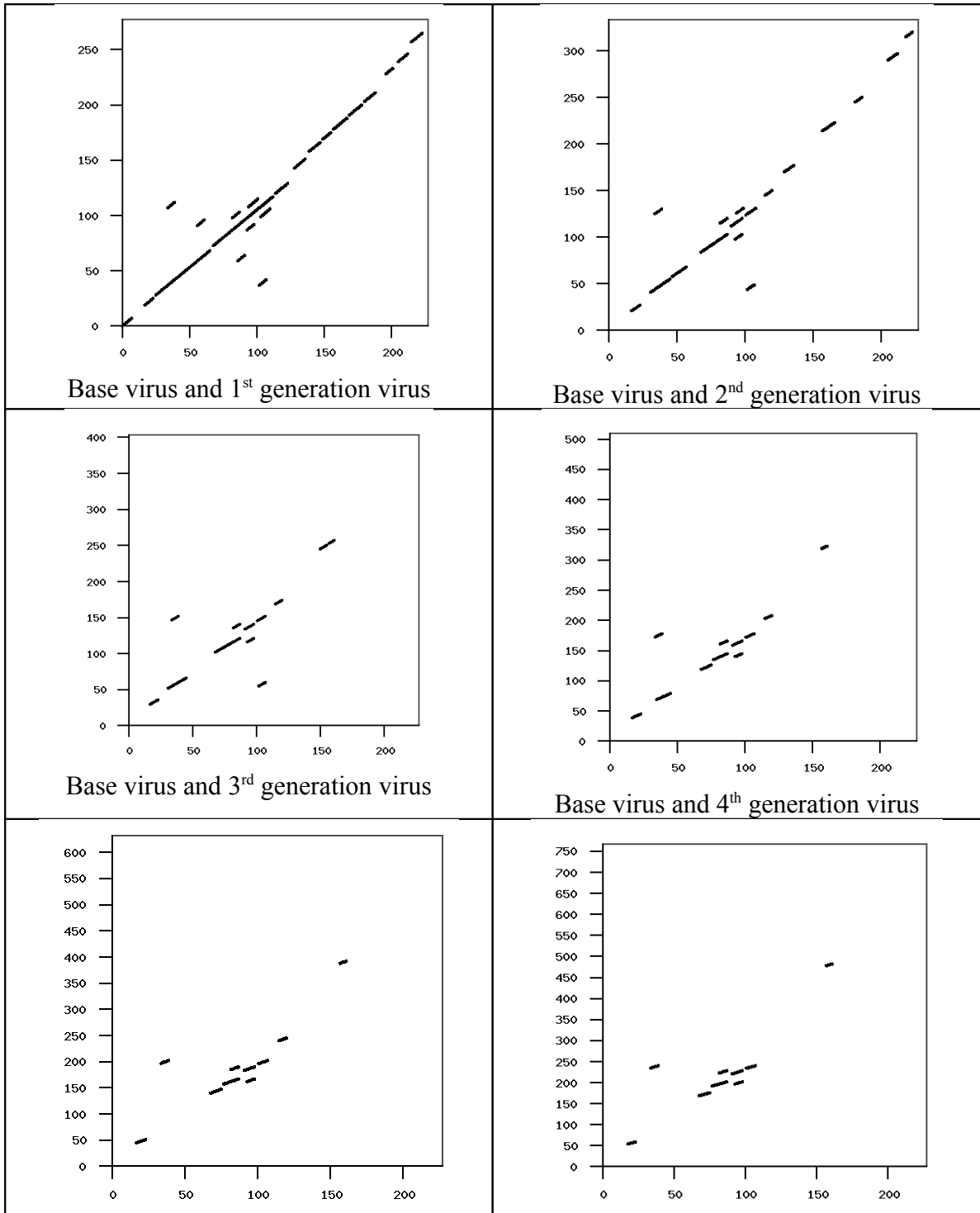
	or R, R2 xor R1, R2 xor R1, R pop RR 2. not R1 not R2 or R1, R2 not R1
dec R	1. neg R not R
dec mem	1. neg mem not mem
inc R	1. add R, 1 2. not R neg R
inc mem	1. add mem, 1 2. not mem neg mem
invoke op1, op2	1. stdcall [op1], op2
jmp loc	1. cmp RR, RR jz loc
jmp R	1. push R ret
lea R, [R1 + R2]	1. mov R, R1 add R, R2
lea R, [R + R1 + imm]	1. add R, imm add R, R1
lea R, [R1 + R2 + imm]	1. lea R, [R1 + imm] add R, R2
lodsb	1. mov al, [esi] add esi, 1
lodsd	1. mov eax, [esi] add esi, 4
movsb	1. push eax mov al, [esi] add esi, 1 mov [edi], al add edi, 1 pop eax
movsd	1. push eax mov [eax], esi add esi, 4 mov [edi], eax

	add edi, 4 pop eax
neg R	1. not R add R, 1
neg mem	1. not mem add mem, 1
not R	1. neg R sub R, 1 2. neg R dec R 3. neg R add R, -1 4. xor R, -1
not mem	1. neg mem sub mem, 1 2. neg mem dec mem 3. neg mem add mem, -1
or R1, R2	1. push RR mov RR, R1 xor RR, R2 and R1, R2 xor R1, RR pop RR
or R1, mem	1. push RR mov RR, R1 xor RR, mem and R1, mem xor R1, RR pop RR
or R1, imm	1. push RR mov RR, R1 xor RR, imm and R1, imm xor R1, RR pop RR
or mem, R	1. push RR mov RR, mem xor RR, R and mem, R xor mem, RR pop RR

or mem, imm	<ol style="list-style-type: none"> 1. push RR mov RR, mem xor RR, imm and mem, imm xor mem, RR pop RR
popad	<ol style="list-style-type: none"> 1. pop edi pop esi pop ebp add esp, 4 pop ebx pop edx pop ecx pop eax
stdcall op1, op2	<ol style="list-style-type: none"> 1. invoke [op1], op2
stosb	<ol style="list-style-type: none"> 1. mov edi, [al] add edi, 1
stosd	<ol style="list-style-type: none"> 1. mov edi, [eax] add edi, 4
sub R, imm	<ol style="list-style-type: none"> 1. add R, new imm where new imm = imm x (-1)
sub mem, imm	<ol style="list-style-type: none"> 1. add mem, new imm where new imm = imm x (-1)
sub R, 1	<ol style="list-style-type: none"> 1. neg R not R
sub mem, 1	<ol style="list-style-type: none"> 1. neg mem not mem
test R1, R2	<ol style="list-style-type: none"> 1. or R1, R2
xchg R1, R2	<ol style="list-style-type: none"> 1. xor R1, R2 xor R2, R1 xor R1, R2
xor R, R	<ol style="list-style-type: none"> 1. sub R, R 2. mov R, 0 3. and R, 0

Appendix C: Similarity Tests

Table C-1: Comparison results of the base virus with N generations



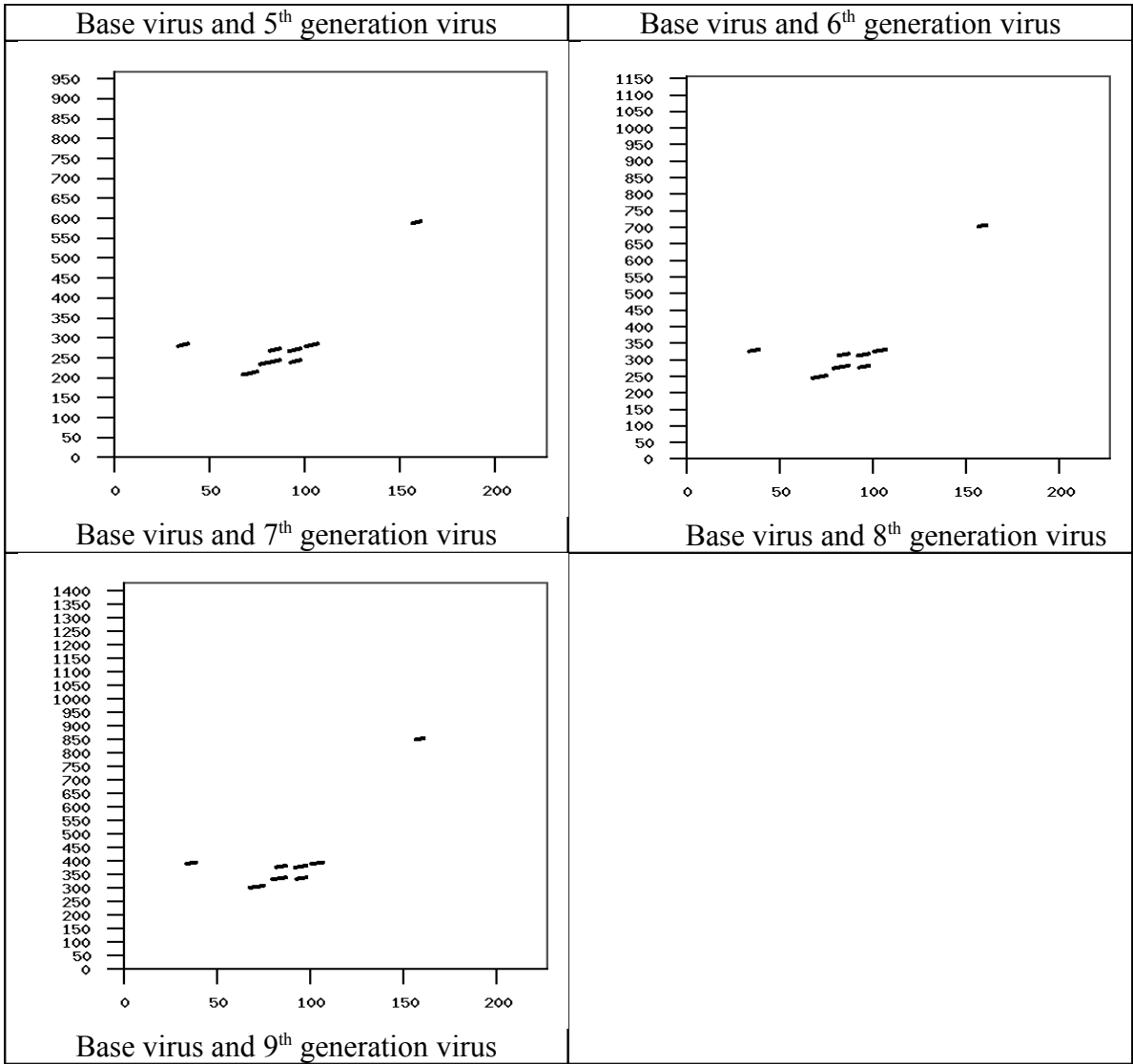
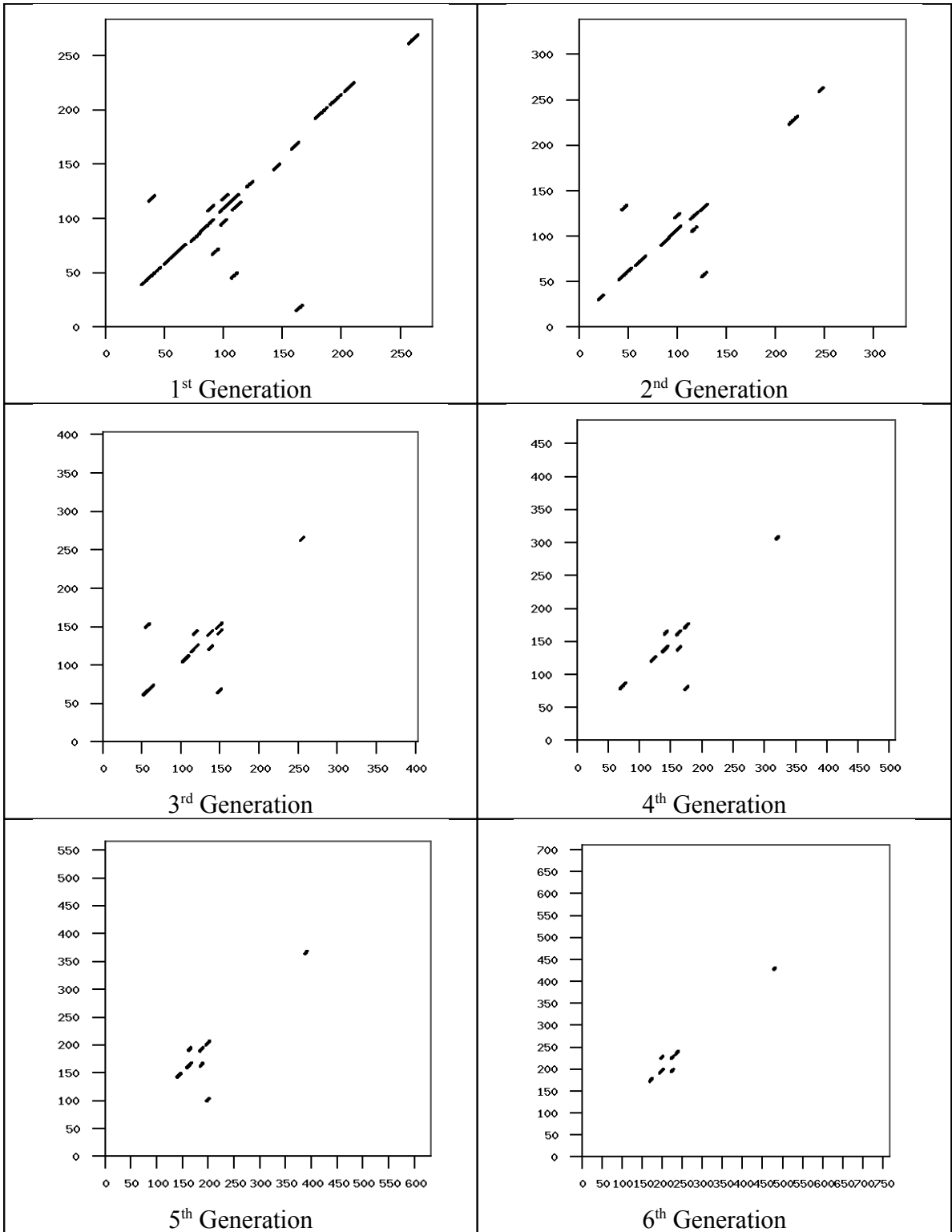
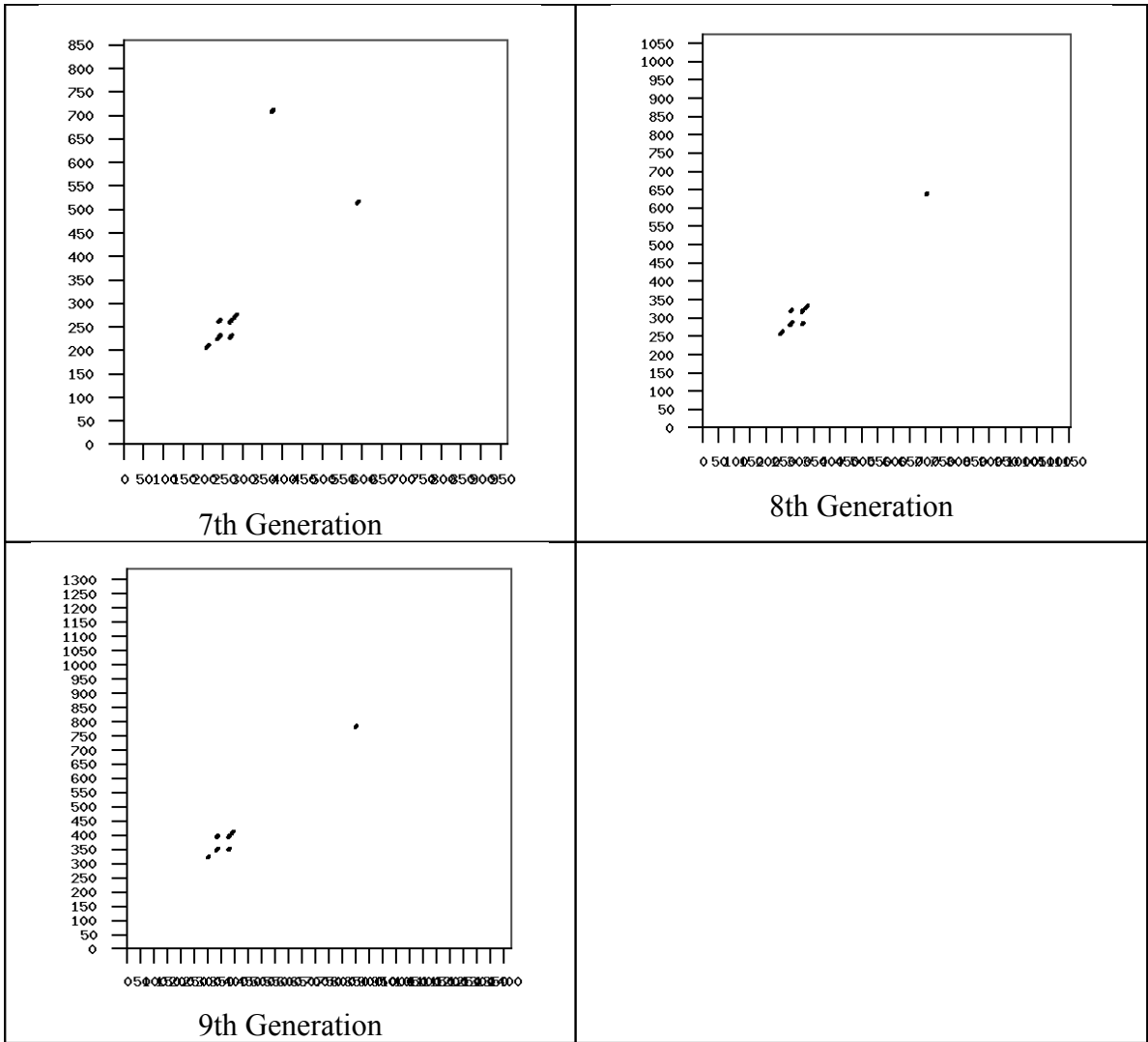


Table C-2: Comparison of N generations





Appendix D: Hidden Markov Model of the Base Virus

Table D-1: HMM parameters (A, B, π) of the base virus with N = 2

N = 2, M = 27, T = 13620		
π :	0.0000000000000000	1.0000000000000000
A:	0.000000000000454	0.999999999999544
	0.78098025609290	0.21901974390710
B:		
start	0.0000000000000000	0.01569168387372
call	0.16075993445552	0.0000000000000000
pop	0.01818103103059	0.04856961642305
sub	0.03658902222441	0.01850358587934
xor	0.01367743341443	0.00501131319206
mov	0.22927558802501	0.22110228244243
lodsd	0.02009499180694	0.0000000000000000
add	0.12595237659940	0.18409720285682
inc	0.0000000000000000	0.03138336774743
cmp	0.0000000000000000	0.10199594517915
jnz	0.08037996722776	0.0000000000000000
dec	0.0000000000000000	0.02353752581057
lea	0.03647826819294	0.01859007097306
push	0.09843010755634	0.22128034835740
stosd	0.0000000000000000	0.00784584193686
lodsb	0.01004749590347	0.0000000000000000
loop	0.0000000000000000	0.00784584193686
test	0.0000000000000000	0.03138336774743
jz	0.10047495903470	0.0000000000000000
movzx	0.0000000000000000	0.02353752581057
imul	0.01004749590347	0.0000000000000000
pusha	0.0000000000000000	0.00784584193686
popa	0.0000000000000000	0.00784584193686
rep	0.0000000000000000	0.01569168387371
retn	0.00937384910765	0.00824111208583
jmp	0.04018998361388	0.0000000000000000
jle	0.01004749590347	0.0000000000000000

Table D-2: HMM parameters (A, B, π) of the base virus with N = 3

N = 3, M = 27, T = 13620			
π :	1.0000000000000000	0.0000000000000000	0.0000000000000000
A:	0.10040502462601	0.08365175778876	0.81594321758522
	0.12520909122804	0.87479090877196	0.0000000000000000
	0.0000000000000000	0.12467619840110	0.87532380159890
B:			
start	0.00108025173100	0.0000000000000000	0.01966848475016
call	0.0000000000000000	0.0000000000000000	0.15867012907693
pop	0.0000000000000000	0.03556280523263	0.04028836621593
sub	0.0000000000000000	0.05419236906398	0.0000000000000000
xor	0.0000000000000000	0.01806412302133	0.0000000000000000
mov	0.0000000000000000	0.41203378876197	0.05336255596307
lodsd	0.0000000000000000	0.01806412302133	0.0000000000000000
add	0.0000000000000000	0.32515421438388	0.0000000000000000
inc	0.0000000000000000	0.01806412302133	0.01983376613462
cmp	0.49078744655348	0.0000000000000000	0.05382769681044
jnz	0.0000000000000000	0.0000000000000000	0.07933506453846
dec	0.0000000000000000	0.0000000000000000	0.02975064920192
lea	0.31368699013508	0.01047971535740	0.0000000000000000
push	0.0000000000000000	0.02709618453199	0.34709090735578
stosd	0.0000000000000000	0.0000000000000000	0.00991688306731
lodsb	0.0000000000000000	0.0000000000000000	0.00991688306731
loop	0.0000000000000000	0.0000000000000000	0.00991688306731
test	0.0000000000000000	0.0000000000000000	0.03966753226923
jz	0.0000000000000000	0.0000000000000000	0.09916883067308
movzx	0.0000000000000000	0.02709618453199	0.0000000000000000
imul	0.0000000000000000	0.00903206151066	0.0000000000000000
pusha	0.06481510386015	0.0000000000000000	0.0000000000000000
popa	0.0000000000000000	0.000000000000821	0.00991688305829
rep	0.12963020772030	0.0000000000000000	0.0000000000000000
retn	0.0000000000000000	0.0000000000000000	0.01966848475016
jmp	0.0000000000000000	0.03612824604265	0.0000000000000000
jle	0.0000000000000000	0.00903206151066	0.0000000000000000

Appendix E: Hidden Markov Models of Normal Files

Table E-1: HMM parameters (A, B, π) for Normal Files with N = 2

N = 2, M = 56, T = 7351					
π :					
1.0000000000000000		0.0000000000000000			
A:					
0.86450620287537		0.13549379712462			
0.04500882863247		0.95499117136753			
B:					
start	0.02837277526002	0.0000000000000000	dec	0.0000000000000000	0.00090624740380
push	0.23166166493378	0.00243304736880	arpl	0.00436504234770	0.0000000000000000
mov	0.18848829422565	0.55363549869642	cld	0.0000000000000000	0.00181249480761
sub	0.00575803807657	0.05663085598714	repe	0.0000000000000000	0.00253749273065
and	0.01273100451890	0.01842715311818	movsx	0.0000000000000000	0.00054374844228
test	0.0000000000000000	0.02537492730648	jg	0.0000000000000000	0.00090624740380
jz	0.0000000000000000	0.03208115809462	inc	0.0000000000000000	0.00471248649977
int	0.0000000000000000	0.00471248649977	setnz	0.0000000000000000	0.00036249896152
fnstcw	0.0000000000000000	0.00489373598054	popa	0.00054563029346	0.0000000000000000
movzx	0.00047405096064	0.01416123698577	outsb	0.00109126058692	0.0000000000000000
or	0.0000000000000000	0.01558745534541	setz	0.0000000000000000	0.00090624740380
fldcw	0.0000000000000000	0.00507498546130	jge	0.0000000000000000	0.00181249480761
call	0.0000000000000000	0.10584969676417	jbe	0.0000000000000000	0.00144999584608
leave	0.03382907819464	0.0000000000000000	shl	0.0000000000000000	0.00072499792304
retn	0.12604059778972	0.0000000000000000	shr	0.0000000000000000	0.00036249896152
cmp	0.0000000000000000	0.03171865913310	neg	0.0000000000000000	0.00018124948076
jle	0.0000000000000000	0.00181249480761	sar	0.0000000000000000	0.00036249896152
xor	0.01057672968058	0.02331150612693	jl	0.0000000000000000	0.00163124532685
lea	0.0000000000000000	0.02555617678724	jns	0.0000000000000000	0.00072499792304
pop	0.15932404569090	0.0000000000000000	cdq	0.0000000000000000	0.00018124948076
jmp	0.12567211288436	0.01860985171079	xchg	0.00109126058692	0.0000000000000000
add	0.0000000000000000	0.02048119132594	ror	0.00054563029346	0.0000000000000000
jb	0.00572386768789	0.00734234806584	js	0.00889051083744	0.00121545541851
jnz	0.0000000000000000	0.00634373182662	ja	0.0000000000000000	0.00163124532685
jnb	0.01225924226040	0.00100266520921	fstp	0.0000000000000000	0.00090624740380
insw	0.01200386645616	0.0000000000000000	fld	0.0000000000000000	0.00072499792304
insb	0.01200386645616	0.0000000000000000	fsub	0.0000000000000000	0.00018124948076
imul	0.01855142997771	0.0000000000000000	fistp	0.0000000000000000	0.00018124948076

Table E-2: HMM parameters (A, B, π) for Normal Files with N = 3

N = 3, M = 56, T = 7351							
π :		0.00000	0.0000	1.00000			
A:		0.11796	0.23553	0.64649			
		1.00000	0.00000	0.00000			
		0.00000	0.10631	0.89368			
B:							
start	0.00916	0.00000	0.00775	dec	0.00000	0.00367	0.00035
push	0.00139	0.00000	0.07778	arpl	0.00000	0.00982	0.00000
mov	0.05227	0.00000	0.59701	cld	0.00000	0.00000	0.00178
sub	0.00000	0.07713	0.04634	repe	0.00000	0.01719	0.00000
and	0.02354	0.00000	0.01839	movsx	0.00108	0.00000	0.00035
test	0.00000	0.17199	0.00000	jg	0.00426	0.00000	0.00018
jz	0.18844	0.00000	0.00058	inc	0.00000	0.00512	0.00388
int	0.02820	0.00000	0.00000	setnz	0.00216	0.00000	0.00000
fnstcw	0.00000	0.03317	0.00000	popa	0.00108	0.00000	0.00000
movzx	0.04580	0.03909	0.00088	outsb	0.00000	0.00245	0.00000
or	0.09083	0.00000	0.00040	setz	0.00541	0.00000	0.00000
fildcw	0.00000	0.00000	0.00498	jge	0.01084	0.00000	0.00000
call	0.00000	0.00000	0.10401	jbe	0.00867	0.00000	0.00000
leave	0.00000	0.07616	0.00000	shl	0.00000	0.00249	0.00035
retn	0.25062	0.00000	0.00000	shr	0.00000	0.00000	0.00035
cmp	0.00000	0.21499	0.00000	neg	0.00000	0.00000	0.00017
jle	0.01084	0.00000	0.00000	sar	0.00000	0.00000	0.00035
xor	0.00116	0.00000	0.02616	jl	0.00976	0.00000	0.00000
lea	0.00000	0.00000	0.02511	jns	0.00254	0.00203	0.00000
pop	0.07111	0.27820	0.00000	cdq	0.00000	0.00000	0.00017
jmp	0.00000	0.00000	0.05931	xchg	0.00000	0.00149	0.00013
add	0.00000	0.00000	0.02012	ror	0.00108	0.00000	0.00000
jb	0.05533	0.00000	0.00000	fstp	0.00108	0.00000	0.00071
jnz	0.03045	0.00000	0.00123	fld	0.00000	0.00000	0.00071
jnb	0.00556	0.02810	0.00000	fsub	0.00000	0.00122	0.00000
insw	0.02386	0.00000	0.00000	fistp	0.00000	0.00000	0.00017
insb	0.00000	0.02702	0.00000	js	0.02495	0.00000	0.00000
imul	0.02859	0.00856	0.00011	ja	0.00976	0.00000	0.00000

Appendix F: Hidden Markov Model of 9th Generation Viruses

Table F-1: HMM parameters (A, B, π) for 9th Generation viruses with N= 2

N = 2, M = 47, T = 87227				
π :		1.0000000000000000	0.0000000000000000	
A:		0.83217007332176	0.16782992667822	
		0.08209140361062	0.91790859638939	
B:				
start	0.00224949833917	0.00197266362902	loop	0.0000000000000000 0.00153657231855
fmul	0.13115095949748	0.01746554640060	jz	0.00052823655886 0.01510730365676
and	0.17276039691696	0.01174130398615	movzx	0.00094894665121 0.00414548110843
shl	0.13727941735826	0.02718683702829	imul	0.0000000000000000 0.00153657231855
sub	0.11372459106253	0.03937598271045	pusha	0.00002523370760 0.00152422769242
fdiv	0.13170570484462	0.01825268605848	rep	0.0000000000000000 0.00307314463711
test	0.02444614244857	0.01960868462539	retn	0.00051071682025 0.00284036899599
shr	0.03304940152925	0.0000000000000000	jle	0.0000000000000000 0.00153657231855
push	0.00713112355913	0.14030039356432	popa	0.0000000000000000 0.00017073025762
mov	0.03714484588762	0.10603456753864	cli	0.0000000000000030 0.00013658420595
pop	0.02277679405086	0.09742469565353	rcr	0.0000000000000000 0.00001707302576
fadd	0.00633423017186	0.02556683045773	retf	0.0000000000000000 0.00010243815457
fsub	0.00735821962914	0.02573173079309	rol	0.00002939919037 0.00003683664806
inc	0.01092985748609	0.01724059842838	fild	0.0000000000000000 0.00003414605152
dec	0.01260818677659	0.01775123603675	jmp	0.02679258523014 0.03806061083191
lea	0.00498067906121	0.01948515838277	fstp	0.00679951925229 0.02609041870823
neg	0.0000000000000000	0.01949739541988	adc	0.01029664573503 0.02427714520424
add	0.02522026658893	0.07540022875981	sbb	0.01075435812487 0.02238007040261
not	0.0000000000000000	0.02011202434730	call	0.0000000000000000 0.02555831956528
lodsd	0.0000000000000000	0.00276583017340	fst	0.00579144863872 0.02817137008812
or	0.02324562742933	0.02031550185330	jnz	0.00030713494533 0.01214232452585
fld	0.00751349266042	0.02435821944460	stosd	0.0000000000000000 0.00153657231855
xor	0.02553306865696	0.02290130532883	lodsb	0.0000000000000000 0.00153657231855
cmp	0.00007327119005	0.01795912402610		

Table F-2: HMM parameters (A, B, π) for 9th Generation viruses with N= 3

N = 3, M = 47, T = 87227							
π :		0.00000	0.00000	1.00000			
A:		0.90141	0.00000	0.09858			
		0.37895	0.08330	0.53774			
		0.00001	0.62948	0.37051			
B:							
start	0.00257	0.00000	0.00212	rcr	0.00001	0.00000	0.00000
fmul	0.02255	0.06837	0.13061	retf	0.00011	0.00000	0.00000
and	0.01882	0.08862	0.16907	rol	0.00005	0.00000	0.00000
shl	0.03193	0.10137	0.12015	fild	0.00003	0.00000	0.00000
sub	0.04047	0.10276	0.09859	not	0.00000	0.06270	0.01536
fdiv	0.02233	0.20430	0.04094	lodsd	0.00263	0.00158	0.00000
test	0.02157	0.01948	0.02138	or	0.02292	0.01813	0.01908
shr	0.00000	0.01738	0.03502	fld	0.02541	0.01759	0.00228
push	0.15554	0.01043	0.00000	xor	0.02201	0.03781	0.01874
mov	0.11757	0.03208	0.02844	cmp	0.01941	0.00150	0.00000
pop	0.10929	0.00000	0.02691	jnz	0.01219	0.00000	0.00353
fadd	0.02761	0.00463	0.00720	stosd	0.00000	0.00000	0.00446
fsub	0.02737	0.01762	0.00085	lodsb	0.00000	0.00649	0.00000
inc	0.00000	0.00000	0.06561	loop	0.00169	0.00000	0.00000
dec	0.00000	0.10118	0.00000	jz	0.01268	0.00000	0.01115
lea	0.02084	0.00000	0.00865	movzx	0.00484	0.00000	0.00060
jmp	0.04327	0.01806	0.02202	imul	0.00167	0.00000	0.00004
fstp	0.02793	0.00427	0.00879	pusha	0.00168	0.00000	0.00002
adc	0.02622	0.01564	0.00520	rep	0.00338	0.00000	0.00000
sbb	0.02370	0.01739	0.00579	retn	0.00340	0.00000	0.00000
call	0.02813	0.00000	0.00000	jle	0.00169	0.00000	0.00000
fst	0.02974	0.01103	0.00397	popa	0.00018	0.00000	0.00000
neg	0.00000	0.00646	0.05219	cli	0.00015	0.00000	0.00000
add	0.06624	0.01300	0.07111				